Imperial College of Science, Technology and Medicine
Department of Electrical and Electronic Engineering

Harnessing Reconfigurable Hardware Capabilities for 3D CNN Acceleration

Petros Toupas

A Thesis submitted in part fulfilment of requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering
of Imperial College London and the Diploma of Imperial College London

September 2025

Abstract

As computer vision research shifts towards video and volumetric data, the demand for efficient processing of spatiotemporal and multi-dimensional data has grown significantly. This thesis investigates the optimisation and mapping of 3D Convolutional Neural Networks (CNNs) onto Field-Programmable Gate Arrays (FPGAs), addressing the growing demand for hardware acceleration in applications like human action recognition (HAR), and medical image segmentation. Compared to 2D CNNs, 3D CNNs introduce additional complexity, larger workloads and higher memory demands. The research explores novel FPGA-based methodologies for both throughput- and latency-optimized designs to address these challenges. Synchronous dataflow is utilised to generate intermediate representations of 3D CNNs, facilitating transformations to expand the design space. A key contribution is the development of automated toolflows for mapping and optimising 3D CNNs onto FPGA devices, addressing the lack of model-agnostic solutions. Adaptive optimization strategies are explored to automatically adjust to varying model architectures and hardware constraints. The research emphasizes on balancing between on-chip and off-chip memory utilisation, as the large data volumes of 3D models often exceed on-chip memory resources requiring off-chip access that can degrade performance. Strategies to address these challenges are investigated, particularly in streaming architectures featuring complex blocks like residual connections and branching. Another key contribution is the introduction of novel design space exploration and performance modelling methods, for efficiently navigating the 3D CNNs' broadened design space. The study additionally investigates runtime parametrisation and reconfigurable computing techniques to improve the flexibility and efficiency of the proposed accelerators. The findings demonstrate significant improvements in both throughput and latency oriented architectures across several 3D CNN models, highlighting the efficiency of the proposed solutions. While primarily focused on Human Action Recognition (HAR), the methodologies of this thesis are broadly applicable to other domains, like video analysis and medical imaging, enabling the 3D CNN implementation and deployment in resource-constrained settings.

Copyright Declaration

© The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC). Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose. When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes. Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Statement of Originality

I hereby declare that this thesis is my own work and comprises my own original research, except where explicitly acknowledged through proper references. Collaborative efforts have been appropriately acknowledged, and this work has not been submitted, in whole or in part, for any degree or qualification at this or any other university.

Petros Toupas
September 2025

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Christos-Savvas Bouganis, for being not only my supervisor but also an incredible mentor. You have shaped the way I think and approach problems as a researcher, and you have also shaped me as an individual. During some difficult moments, you were always there to offer support and advice, helping me overcome every obstacle. I am truly thankful for everything you have done for me. I also want to thank you for giving me the opportunity to travel and meet so many wonderful people around the world. Those experiences have been life-changing.

I would also like to thank my co-supervisor, Dr. Dimitrios Tzovaras, for your help and support over the years. I would not have had the chance to pursue this PhD without you, and I deeply appreciate all you have done to make it possible.

One of the most valuable lessons I learned during this journey is how rewarding collaborations can be. Collaborating with others not only leads to the best results but also makes the journey both enjoyable and inspiring. I am especially grateful to my friends and colleagues, Alex and Zhewen, who are not only brilliant people but also a constant source of support and inspiration. Our discussions and brainstorming sessions have been eye-opening and have helped me grow in so many ways. It has been a pleasure working with you both, thank you! I would also like to thank Giorgos, Ben, Omar, and everyone else I have met in the CAS research group. It has been a great experience getting to know you and learning from our conversations.

I am deeply thankful to my two best friends, Dimitris and Nikos, who have always stood by me. Dimitris, during the first two demanding years of my PhD, you were my flatmate and a constant source of support. I can't thank you enough for being there for me in every way possible. Nikos, your advice and unique perspective have guided me through so many challenges, and I deeply value your constant encouragement and guidance. Thank you both for your immense support throughout this journey.

Finally, I want to express my deepest gratitude to my parents, Christos and Mairi, and my sister, Giota. Your belief in me and your constant support have meant everything

to me. I would not have been able to achieve this without you and the sacrifices you have made for me. Thank you all from the bottom of my heart.

Contents

Abstract i

Copyright Declaration iii

Statement of Originality v

Acknowledgements vii

List of figures xiii

List of tables xix

Nomenclature xxi

1 Introduction 1

1.1 Motivation . . . 3
1.2 Contribution . . . 4
1.3 Thesis Overview . . . 5
1.4 Published Work . . . 7

2 Background 9

2.1 Deep Learning and AI . . . 10
2.1.1 Convolutional Neural Networks (CNNs) . . . 12
2.1.2 2D vs 3D Convolution . . . 17
2.1.3 CNN Training and Inference . . . 27
2.2 AI Hardware Accelerators . . . 28
2.2.1 Application-Specific Integrated Circuits (ASICs) . . . 30
2.2.2 Graphics Processing Units (GPUs) . . . 30
2.2.3 Field-Programmable Gate Arrays (FPGAs) . . . 31

2.3 Human Action Recognition . . . 40
2.3.1 Human Action Understanding . . . 41
2.3.2 Architecture Styles for HAR . . . 42
2.3.3 Evaluation Process and Public Benchmarks . . . 45
2.4 Overview of 3D CNN HAR Accelerators . . . 48
2.4.1 Early Efforts and Advanced Optimisation Strategies . . . 48
2.4.2 Limitations of Existing Approaches . . . 50
2.4.3 Contributions of This Thesis . . . 51

3 Foundations of 3D CNN Acceleration 53
3.1 Introduction . . . 53
3.2 Background . . . 56
3.3 FPGA-based Modelling of 3D CNNs . . . 59
3.3.1 DAG Representation . . . 60
3.3.2 Converting to Hardware Building Blocks . . . 62
3.3.3 SDFG Representation . . . 64
3.4 Design Space Exploration and Optimisation . . . 65
3.4.1 Model Partitioning . . . 66
3.4.2 Performance and Resource Modelling . . . 67
3.4.3 Partition Level Optimisations . . . 69
3.4.4 Model Specific Optimisations . . . 74
3.5 Performance Evaluation and Results . . . 76
3.5.1 Experimental Setup . . . 76
3.5.2 Quantisation Experiments . . . 77
3.5.3 Performance Evaluation . . . 78
3.6 Conclusions . . . 83

4 The fpgaHART Toolflow 85
4.1 Introduction . . . 85
4.2 Background . . . 87
4.3 fpgaHART Toolflow . . . 90
4.3.1 Extra DAG Nodes and Hardware Building Blocks . . . 92
4.3.2 Partitioning Strategies . . . 93
4.4 Performance Evaluation and Results . . . 101
4.4.1 Experimental Setup . . . 101
4.4.2 Performance Evaluation . . . 101

4.5 Conclusions . . . 109

5 Off-Chip Memory Optimisation Strategies 111
5.1 Introduction . . . 111
5.2 Background . . . 114
5.3 Methodology . . . 117
5.3.1 Activation Eviction . . . 117
5.3.2 Weight Fragmentation . . . 119
5.3.3 Partition Reconfiguration . . . 121
5.4 Implementation . . . 123
5.4.1 System Architecture . . . 123
5.4.2 DSE and Optimisation . . . 125
5.4.3 Pipeline Depth Estimation . . . 128
5.5 Performance Evaluation and Results . . . 131
5.5.1 Baseline Setup . . . 131
5.5.2 Off-Chip Streaming . . . 133
5.5.3 Off-Chip Streaming Compression . . . 134
5.5.4 Comparison with related FPGA works . . . 136
5.6 Conclusions . . . 140

6 The HARFLOW3D Toolflow 143
6.1 Introduction . . . 143
6.2 Proposed Architecture . . . 145
6.2.1 Building Blocks Description . . . 146
6.2.2 Hardware Design and Implementation . . . 148
6.3 Modelling . . . 152
6.3.1 Performance Modelling . . . 152
6.3.2 Resource Modelling . . . 154
6.4 Latency-Driven Design Space Exploration . . . 155
6.4.1 Scheduling . . . 156
6.4.2 Optimisation Strategy . . . 157
6.4.3 Transformations . . . 159
6.5 Performance Evaluation and Results . . . 162
6.5.1 Hardware Model Validation . . . 162
6.5.2 Performance Evaluation . . . 163
6.6 Conclusions . . . 170

7 Conclusions **173**
7.1 Summary of Contributions 173
7.2 Future work 176
7.3 Final Thoughts and Reflections 178

References **181**

List of figures

2.1 A single-hidden-layer FNN comprising 128 neurons. 12
2.2 A simple CNN model comprising the three fundamental, Convolution, Pooling, and Nonlinear Activation layers. 14
2.3 Visual representation of the Convolution in various configurations of the main parameters: Kernel Size, Stride, Padding, and Dilation. 18
2.4 Operation of 2D convolution on single channel input feature map 21
2.5 Operation of 2D convolution on multiple channel input feature map . . . 22
2.6 Operation of 3D convolution on single channel input feature map 24
2.7 Comparing ResNet Model Characteristics and Statistics between 2D and 3D Versions 25
2.8 Comparing ShuffleNetv2 Model Characteristics and Statistics between 2D and 3D Versions 25
2.9 Comparing SqueezeNet Model Characteristics and Statistics between 2D and 3D Versions 26
2.10 Comparing UNet Model Characteristics and Statistics between 2D and 3D Versions 26
2.11 Single engine architecture, in the form of a systolic array, for FPGA-based DNN acceleration, featuring a grid of Processing Elements (PEs), on-chip buffers for weights, feature maps, and outputs, communicating with the off-chip memory for data loading and offloading. 35
2.12 Streaming architecture for FPGA-based DNN acceleration. The pipeline processes data across dedicated hardware blocks optimized for individual DNN layers (e.g., convolution, activation, pooling), leveraging layer-specific parallelism. Weights are managed in on-chip memory, while off-chip memory handles the data transfer to and from the first and last layers. . 37

2.13 The flow starts with a high-level description representation of the neural network model, converted to an intermediate DAG representation (IR). The IR is mapped to a hardware-level SDFG, with Design Space Exploration (DSE) optimizing hardware configurations based on platform constraints and specifications. The process generates synthesisable Vitis HLS code for FPGA deployment. 39
2.14 The Pareto front of Top-1 Accuracy (%) versus GFLOPs, over the years (represented by four coloured lines) on Kinetics-400 benchmark. 44
2.15 The Pareto front of Top-1 Accuracy (%) versus GFLOPs, over the years (represented by four coloured lines) on Something-Something-V2 benchmark. 45
2.16 The process of converting raw video data into a structured input format for 3D CNN models. The Figure illustrates the process of both temporal and spatial clipping that form the final input clip to the model. 46
2.17 Pareto front of Top-1 Accuracy (%) versus Throughput (clips/s) for FPGA-based 3D CNN accelerators on the UCF101 [1] HAR benchmark. Marker colours indicate the 3D CNN models targeted in each work, while their shapes represent the FPGA platforms used. The C3D model used by Shen et al. [2] is a different and smaller version from the original one [3] used by the rest of the works. 50

3.1 Pareto front (performance over number of parameters) on the Kinetics-400 benchmark. The lower left side of the graph demonstrates that 3D CNNs achieve Pareto front solutions having orders of magnitude less parameters than Vision Transformer architectures. 55
3.2 Pareto front (performance over number of operations) on the Kinetics-400 benchmark for 3D CNNs. 56
3.3 X3D model main building blocks consisting of a series of 3D convolutions, 3D batch normalisation layers, non-linear activations, and specialised blocks like Squeeze and Excitation and Swish activations. 57
3.4 The basic X3D model block types. We identified and defined these block types based on their structure and receptiveness inside the X3D model architecture. 75
3.5 Illustration of the proposed streaming-centric approximation on GAP layers for improving the throughput of X3D. 76

3.6 Quantisation results on UCF101 over different word lengths. Wx denotes the word length of the fixed point representation of the weights, while FMx for the feature maps 78
3.7 PE on throughput (clips/s) between modelling and co-simulation results. 79
3.8 Positioning of FFM-X3D on the Pareto Front of Accuracy vs. Clips per Second. 80

4.1 Architecture variations across three 3D CNN architectures-C3D, R(2+1)D, and SlowOnly (left to right), showcasing the different layer connectivity and operation types used in each model. The notation $W < F \times C \times D \times H \times W$ in 3D convolutional layers represents the convolutional kernel where F is the number of filters, C is the number of channels, while D, H, W are the depth, height, and width dimensions of the kernel. 89
4.2 High level overview of the fpgaHART toolflow architecture. The three distinct but connected parts are being shown in the diagram. The entry point which is the model parser creates the intermediate representation (IR) and feeds that to the optimiser. During the DSE, the optimiser exchange information with the resource and performance models. Once the optimisation is finished the automated hardware mapping tool converts IR into HDL code and then into a bitstream. 90
4.3 Partitioning the model graph after a branch merging point. The branches remain intact, with a FIFO buffer added to the higher-rate branch to synchronize dataflow and prevent pipeline deadlocks. 97
4.4 Partitioning before the branch merging point or mid branch generates two partitions: one multi-output sequential partition and one multi-input partition that includes the merging node. This approach reduces on-chip memory usage by avoiding extra buffering, while the multi-input, multi-output partitions require careful management of memory bandwidth due to multiple connections to off-chip memory. 98
4.5 Partitioning at bottleneck layers directs output data straight to off-chip memory for the next partition. This strategy helps bypass performance bottlenecks but requires managing the memory demands of multi-input and multi-output partitions. 99

4.6 Overall execution timeline for a model divided into P partitions. The total latency is the sum of each partition's execution latency and the reconfiguration time between partitions. 100
4.7 Resource availability across different FPGA platforms: (a) Comparison of DSPs and BRAMs, showcasing the computational and memory capabilities; (b) Comparison of FFs and LUTs, highlighting the logic resources available.102
4.8 Throughput (GOPs/s) of fpgaHART-generated designs on 3D CNN HAR models delivering high-throughput results on a variety of FPGA devices . 103
4.9 Pareto front on 3D CNNs: Throughput (Clips/s) over Top-1 Accuracy (%). The fpgaHART results were taken using the VC709 FPGA board, delivering solutions on the Pareto front. 105

5.1 Model structures of ResNet50 on the ImageNet classification task [4] and UNet on the CamVid segmentation task [5]. These two models are similar in the number of weight parameters. However, UNet requires much more computation effort due to a high input resolution and its lengthy and hierarchical branch connections pose a unique challenge for streaming architectures. 112
5.2 Activation eviction. For the long skip connection, instead of being held in the on-chip buffer, the activation data is pushed to off-chip memory. We support lossless encoding schemes, such as RLE and Huffman, to save the off-chip bandwidth. 118
5.3 Weight fragmentation. The original weights with depth of d_w is fragmented into static and dynamic regions. Weights in the dynamic regions are loaded from the off-chip memory at runtime, sharing the same piece of physical memory space in a time-multiplexed manner. The ratio of dynamic regions is denoted as m. 120
5.4 Graph manipulation strategies. Figure 5.4a showcases the graph partitioning strategy into multiple partitions for different reconfiguration points. In Figure 5.4b the off-chip streaming strategy is presented, utilising the activation eviction method. By streaming the long skip connections to off-chip and reading back from it on merge points the graph can fit into the device without needing multiple partitions and hence device reconfiguration.122

5.5 A streaming accelerator design example with the proposed memory optimisation methodology, where the weights of the layer `Conv_22` are partially offloaded to off-chip, and the long skip connection between `Relu_3` and `Concat_47` is also evicted to off-chip. 124
5.6 A segment of the design's waveform illustrates the distinction between the initiation rate r^{st}, which is applicable to the pipeline depth region of the layers, and the input rate r^{in}, evident for the remainder of the layer's execution. 130
5.7 Evaluation of the UNet and UNet3D under four different steaming strategies: *i*) No off-chip streaming allowed (blue), *ii*) Only activations off-chip streaming allowed (orange), *iii*) Only weights off-chip streaming allowed (green), *iv*) Both weights and activations off-chip streaming (red). 133
5.8 Evaluation of the UNet and UNet3D under three compression strategies for off-chip streaming. The off-chip streaming configuration is fixed to *weights & activations streaming*. The different colours denote the encoding scheme adopted: *No encoding*, *Huffman encoding*, and *Run-Length Encoding (RLE)*.135

6.1 Pareto front on 3D CNNs: Latency over Accuracy. Designs produced by the proposed HARFLOW3D toolflow dominate the Pareto front. 144
6.2 High level overview or the HARFLOW3D toolflow architecture. The four distinct but connected parts are being shown in the diagram. The entry point which is the model parser creates the intermediate representation (IR) and feeds that to the optimiser and the scheduler. During the DSE, the optimiser and scheduler, exchange information with the resource and performance models. Once the optimisation is finished the automated hardware mapping tool converts IR into HDL code and then into a bitstream.146
6.3 Block diagram of an accelerator instance produced by the HARFLOW3D toolflow. The black lines describe AXI-Stream signals, where the arrows indicate the directionality of the connection, blue are high-throughput AXI interfaces for DMA access, red are AXI-Lite connections for runtime configuration of the hardware nodes, and green indicate the DDR IO interfaces for communicating with off-chip memory. 149

6.4 Diagram of hardware for Convolution, and how it can be used with runtime parameters. The blue blocks represent compile-time configurable hardware modules. The red blocks represent runtime configurable hardware modules. Cross-hatching gives an example of how hardware elements can be bypassed at runtime. 150
6.5 Evolution of latency during Simulated Annealing for various FPGA devices. 158
6.6 The Dataflow of a simple design consisting of Convolution, ReLU, and FC layers. As the red lines and the crossbar dictate the flow between Convolution and ReLU can be addressed within the FPGA without sending the data back to the off-chip memory. This is the result of Fuse Activation optimisation. 160
6.7 Comparison of predicted latency to measured latency for all the convolution layers of C3D on the ZCU106 board, depicted as percentage error. . . . 164
6.8 Pareto front of DSP utilisation against latency for the R(2+1)D-34 model on a ZCU102 platform during Simulated Annealing optimisation. 166
6.9 Comparison of the DSP efficiency between HARFLOW3D and previous works. The wide range of the supported FPGA platforms allows for a direct comparison. We demonstrate competitive, and in some cases superior performance. 167

List of tables

2.1 Characteristics and statistics on modern CNN architectures between their 2D and 3D versions 27

3.1 Variations of the X3D model evaluated on the K400-val dataset using 10-centre clip testing. Original table information from [6]. 57
3.2 Performance comparison versus GPU 81
3.3 Positioning of FFM-X3D in the space of 3D CNN HAR models 82

4.1 Comparison of 3D CNN model support in fpgaHART versus related works, highlighting the range and diversity of supported architectures. 87
4.2 3D CNN models characteristics 88
4.3 Detailed runtime profile of the fpgaHART-generated C3D accelerator on the ZCU102 platform. The table provides a partition-level breakdown of key performance metrics and illustrates the effect of different batch sizes on total latency and throughput, highlighting the amortisation of reconfiguration overhead. 104
4.4 Comparison with existing works on 3D CNN HAR models 108

5.1 Comparison of our work with related works. 116
5.2 Characteristics of the evaluated CNN models. 131
5.3 Breakdown and impact of model partitioning with device reconfiguration on UNet3D designs targeting different batch sizes. 132
5.4 Resource overheads for using Huffman and RLE schemes. The overhead is applied per data stream containing 8-bit data. For compressing parallel data streams, the overhead will be multiplied by the number of data streams as well. 136

5.5 Comparison of FPGA accelerator architectures generated by SMOF with prior state-of-the-art works on four major computer vision tasks (segmentation, detection, 3D segmentation, action recognition). 139

6.1 Compile time parameters for node n in the hardware graph G, for each layer type. 151
6.2 Comparison of predicted and synthesised resources for C3D designs on a ZCU102 board. 162
6.3 Statistical resource modelling information over multiple synthesis runs for different convolution layers and configurations. 163
6.4 Comparison of HARFLOW3D with existing works, for 3D CNN HAR models. Our tool can target all models, including X3D. 168
6.5 Comparison of HARFLOW3D against Nvidia RTX 2060 GPU on the C3D model. 169

Nomenclature

Acronyms / Abbreviations

2D	Two Dimensional
3D	Three Dimensional
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BFP	Block Floating Point
BRAM	Block RAM
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DDR	Double Data Rate
DNN	Deep Neural Network
DSE	Design Space Exploration
DSP	Digital Signal Processor
FC	Fully Connected
FF	Flip-Flop
FIFO	First-In First-Out
FLOP(s)	Floating-Point Operation(s)

FLOPS	Floating Point Operations Per Second
FNN	Feedforward Neural Network
FPGA	Field Programmable Gate Array
Gbps	Gigabits per second
GPU	Graphics Processing Unit
GPU	Graphics Processing Unit
HAR	Human Action Recognition
HDL	Hardware Description Language
HLS	High Level Synthesis
IR	Intermediate Representation
LUT	LookUp Table
MB	MegaByte
NLP	Natural Language Processing
NN	Neural Network
OP(s)	Operation(s)
OPS	Operations Per Second
PE	Processing Element
RLE	Run-Length Encoding
SA	Simulated Annealing
SDFG	Synchronous DataFlow Graph
SDF	Synchronous DataFlow
SE	Squeeze and Excitation
SGD	Stochastic Gradient Descent
URAM	UltraRAM

Chapter 1

Introduction

Artificial Intelligence (AI) has rapidly evolved from a concept limited to research labs and niche applications to a prominent part of our everyday lives. Its presence grows stronger with each passing day, influencing how people engage with technology and reshaping industries. In recent years, AI has emerged as a go-to solution across various domains, integrating into both personal and professional lives. AI-powered technologies have become prevalent in daily life, improving user experiences through tools like computer vision algorithms in smartphone cameras, enabling functionalities such as facial recognition, augmented reality filters, and real-time scene detection. Perhaps more importantly, AI has accelerated transformation in industrial settings, driving innovation in robotics, automotive, security systems, medical diagnostics, and agriculture. These applications highlight AI's versatility and capacity to excel at a variety of tasks. In addition, the impact of AI have affected the focus of research and development in the computer science community. The advancements in AI has necessitated the development of new methodologies and tools, from software frameworks to hardware architectures, which enabled a complementary evolution in both domains. AI continues to push technological limits, transforming not only the tools we use, but also the fundamental approaches we use to solve challenging problems.

The abundance and ever-increasing availability of data have been a critical factor in the remarkable evolution of AI [7, 8]. At the heart of AI lies Machine Learning (ML), a subset of AI that focuses on algorithms enabling machines to learn from data. Machine learning algorithms rely heavily on training data to effectively learn underlying patterns and features, performing a wide range of tasks. The larger the volume and diversity of data, the more opportunities there are for these algorithms to improve their performance, adapt to complicated settings, and attain greater accuracy. Within ML,

Deep Neural Networks (DNNs) have emerged as a powerful approach for leveraging this vast amount of data, with their layered architecture enabling the extraction of patterns and representations from large datasets. The advancements in DNN architectures have empowered these models to scale effectively, enabling them to process and utilize the vast amounts of data required for modern AI applications. Prime examples of such architectural innovations are the ResNet [9] and Transformer [10] architectures, which have been instrumental in driving this shift. This synergy between data availability and architectural innovation is one of the cornerstones for AI's rapid advancement and expanding capabilities.

Another important factor driving AI advancements is the continuous improvement in performance and computing power of the underlying hardware on which AI algorithms are executed. This revolution started with the pioneering work of A. Krizhevsky et al. [11], who utilised GPUs for the first time to train a deep Convolutional Neural Network named AlexNet on an large-scale dataset. This significantly decreased training times and enabled them to outperform competitors in the ImageNet challenge (ILSVRC) [12] by a significant margin (9.8*pp*), marking a pivotal moment in AI history. From that point on, GPUs have become the go-to choice for AI applications, providing the computational horsepower necessary to support increasingly complex models. More recently, Google's Tensor Processing Unit (TPU) [13] have advanced the area by providing specialised hardware optimised for deep learning workloads. Today, the evolution continues with the emergence of domain-specific Application-Specific Integrated Circuits (ASICs), such as the Language Processing Unit (LPU) from Groq [14], which was specifically designed to meet the immense requirements of large language models (LLMs). Other key players have also introduced novel hardware, such as SambaNova Systems with their Reconfigurable Dataflow Unit (RDU)[1] and Tenstorrent with their Tensix Neo[2] architecture, further diversifying the AI hardware landscape. Field-Programmable Gate Arrays (FPGAs) have emerged as a flexible and energy-efficient alternative for AI workloads [15, 16]. Offering a unique balance between the general-purpose parallelism of GPUs and the specialization of ASICs, FPGAs offer a unique middle ground enabling custom hardware configurations tailored to specific AI models. This reconfigurability allows them to adapt to evolving architectures while maintaining efficiency, making them well-suited for edge AI and exploratory research. These hardware breakthroughs, together with algorithmic

[1] https://sambanova.ai/products/sn40l-rdu-ai-chip
[2] https://tenstorrent.com/en/ip/tensix-neo

and data-driven advancements, have been instrumental in AI's explosive growth and expanding capabilities.

1.1 Motivation

The rapid evolution of machine learning (ML) model architectures not only brings structural and architectural changes in the models, but it can also increase their computational and memory requirements. As these models introduce novel primitives and increase in complexity, existing hardware, which often conforms to a "one-size-fits-all" design approach, struggles to keep pace. This general-purpose approach results in inefficiencies, as the hardware may be underutilised for certain workloads, leading to suboptimal performance. The development of new domain-specific hardware to facilitate new model architectures is inherently slow, given the long design and manufacturing cycles for custom chips. This lag poses a significant challenge in aligning hardware capabilities with the ever evolving landscape of machine learning model architectures, highlighting the necessity for more flexible solutions.

To bridge this gap and minimise hardware underutilization while ensuring optimal performance for ML models, the need for solutions that apply model-specific customizations has become essential. Tailoring the workload of machine learning models to the specific hardware can significantly boost performance and efficiency. A range of tools have been developed to streamline this process, including Intel OpenVINO [17, 18], AMD Vitis AI [19, 20], and Nvidia TensorRT [21, 22]. These frameworks enable software and compiler-level optimizations tailored to specific pairs of models and hardware configurations, primarily focusing on these aspects while only partially addressing hardware architecture-level customizations. This leaves a gap for solutions that focus on hardware-level optimizations to meet the specific demands of emerging ML architectures, unlocking new possibilities for efficiency and innovation in AI acceleration.

While numerous works have focused on hardware-level optimizations for FPGAs, proposing solutions tailored to the specific requirements of two-dimensional (2D) Convolutional Neural Network (CNN) models [23–26]. These models are a cornerstone of modern computer vision, excelling at processing grid-like data such as static images. In contrast, the field of three-dimensional (3D) CNNs—which extend these principles to handle volumetric or spatiotemporal data like videos and medical scans—still remains relatively unexplored, with only a handful of studies addressing their unique challenges. Considering the importance of 3D CNN applications like Human Action Recognition

(HAR), medical imaging, and video analysis, which play pivotal roles in domains such as healthcare, surveillance, and autonomous systems, this leaves a significant gap in providing optimized and efficient solutions tailored to their unique characteristics and computational demands. The additional computational and memory demands of 3D CNNs, stemming from their ability to process and handle higher-dimensional spatiotemporal data, demand bespoke solutions rather than adaptations of the already existing methods for 2D CNNs, which often result in inefficiencies or suboptimal performance. Furthermore, the lack of generic, end-to-end tools for optimising 3D CNNs on reconfigurable hardware creates also a notable gap in the field. This thesis aims to fill these gaps by developing efficient, scalable, and model-agnostic tools specifically designed for 3D CNNs, enabling their full potential in impactful applications such as HAR.

1.2 Contribution

The primary contribution of this thesis is a novel methodology for designing and optimizing FPGA-based accelerators for 3D CNNs. Our central insight is that by systematically expanding the design space with new, configurable dimensions—such as automated model partitioning and flexible on-chip/off-chip memory management—we create a richer solution space to explore. Ultimately, the success of this approach is not attributable to a single architectural invention, but in its ability to enable the discovery of superior hardware configurations by systematically and effectively searching this larger design space with automated Design Space Exploration (DSE). This methodology, and its implementation in two novel toolflows, leads to the following specific contributions:

- **Development of strategies to optimize 3D CNN execution on FPGAs**: This research introduces novel techniques aimed at overcoming the computational and memory challenges associated with 3D CNNs. This includes strategies for the efficient management of the vast amounts of intermediate data generated during 3D CNN processing, specifically emphasising sub-graph partitioning algorithms. These solutions allow for the efficient management and processing of the models' workloads, ensuring the adaptability across diverse model architectures and FPGA platforms, for applications such as human action recognition, medical imaging, and video analysis.

- **Design Space Exploration (DSE) and performance/resource models for 3D CNNs**: Specialized Design Space Exploration (DSE) solutions tailored to the

distinctive characteristics of 3D CNNs are introduced in this thesis. Alongside these, precise performance and resource models are developed, providing accurate predictions of hardware resource utilization and implementation's performance. These tools are designed to guide the optimization processes while addressing the unique demands of 3D CNN workloads.

- **Introduction of two toolflows designed for FPGA-based acceleration of 3D CNNs**: This research has resulted into two distinct, end-to-end toolflows designed specifically for 3D CNNs: A throughput-oriented toolflow named fpgaHART[3] and a latency-oriented toolflow named HARFLOW3D[4]. These toolflows integrate the proposed DSE solutions and performance and resource modelling, providing automated solutions capable of generating optimal hardware configurations targeting specific pairs of models and FPGA platforms. Through these toolflows, this thesis explores and evaluates various modern 3D CNN architectures, demonstrating their adaptability and efficiency across a wide range of FPGA platforms. Both toolflows have been made open-source under the GPL-3.0 license to facilitate further research and development in the field.

1.3 Thesis Overview

This thesis is organized into the following chapters, each addressing key aspects of my research:

- **Chapter 2** provides an in-depth discussion of foundational concepts in deep learning and artificial intelligence, with particular focus on convolutional neural networks (CNNs) and the distinctions between 2D and 3D CNNs. It also reviews various AI hardware accelerators, including systolic arrays and streaming architectures. Human Action Recognition (HAR), a very important application of 3D CNNs, is thoroughly discussed, highlighting its significance, architecture styles, evaluation methods, and benchmarks. The chapter concludes with an overview of existing FPGA-based 3D CNN HAR accelerators, setting the stage for the hardware-based optimizations explored in later chapters. a lightweight and efficient architecture designed for high-performance video processing tasks

[3]https://github.com/ptoupas/fpgahart
[4]https://github.com/ptoupas/harflow3d

- **Chapter 3** presents the theoretical and practical aspects for FPGA-based acceleration on 3D CNNs, targeting the X3D model, a state-of-the-art, lightweight and efficient architecture designed for video-related tasks such as Human Action Recognition (HAR). This chapter covers the FPGA-based modelling of 3D CNNs using directed acyclic graph (DAG) representations, as well as the conversion of these intermediate representations (IR) into hardware building blocks utilising synchronous dataflow graphs (SDFGs). Additionally, it discusses key design space exploration and optimization strategies, including model partitioning and performance modelling, and concludes with an evaluation of quantisation experiments and overall performance.

- **Chapter 4** presents fpgaHART, a throughput-oriented toolflow for automating the deployment and optimization of 3D CNNs on FPGAs. This chapter details the development of fpgaHART, which supports multiple models and FPGA platforms and incorporates an automated model-partitioning strategy. This partitioning identifies optimal reconfiguration points for per-partition optimizations, overcoming the scalability limitations of the manual tuning of partitions. fpgaHART is evaluated across various FPGA platforms and 3D CNN architectures, demonstrating its adaptability and high throughput, thus establishing a new standard for, FPGA-based 3D CNN deployment in throughput-oriented applications.

- **Chapter 5** tackles the limitations of traditional streaming dataflow architectures in handling large and complex CNNs. This chapter introduces memory optimization techniques that utilizes both on-chip and off-chip memory in streaming architectures, enabling the acceleration of larger networks. Key contributions include the introduction of activation eviction, and weight fragmentation, a Design Space Exploration (DSE) strategy with special focus on model partitioning and memory utilisation.

- **Chapter 6** introduces HARFLOW3D, a latency-oriented toolflow for deploying 3D CNNs on FPGA platforms. HARFLOW3D comprises a model parser, resource and performance models, a scheduling algorithm, and a resource-aware optimization engine. Through an array of transformations tailored to each model layer, HARFLOW3D achieves significant latency reductions, leveraging parametrised building blocks for runtime flexibility. The toolflow's performance is evaluated across multiple 3D CNN and FPGA configurations, achieving state-of-the-art per-

formance, and marking a substantial advancement in latency-oriented FPGA-based 3D CNN acceleration.

- **Chapter 7** summarizes the findings and contributions of the research, emphasizing the advancements in FPGA-based 3D CNN acceleration achieved through throughput- and latency-oriented designs. This chapter also discusses potential directions for future work in hardware-accelerated deep learning.

1.4 Published Work

The research conducted for this thesis has led to several conference publications, which are listed below.

1. **P. Toupas**, C.-S. Bouganis and D. Tzovaras, "FMM-X3D: FPGA-Based Modeling and Mapping of X3D for Human Action Recognition," 2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP), Porto, Portugal, 2023, pp. 119-126. This publication is relevant to **Chapter 3**.

2. **P. Toupas**, C.-S. Bouganis and D. Tzovaras, "fpgaHART: A Toolflow for Throughput-Oriented Acceleration of 3D CNNs for HAR onto FPGAs," 2023 33rd International Conference on Field-Programmable Logic and Applications (FPL), Gothenburg, Sweden, 2023, pp. 86-92. This publication is relevant to **Chapter 4**.

3. **P. Toupas***, A. Montgomerie-Corcoran*, C.-S. Bouganis and D. Tzovaras, "HARFLOW3D: A Latency-Oriented 3D-CNN Accelerator Toolflow for HAR on FPGA Devices," 2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Marina Del Rey, CA, USA, 2023, pp. 144-154. This publication is relevant to **Chapter 6**.

4. A. Montgomerie-Corcoran, **P. Toupas**, Z. Yu and C.-S. Bouganis, "SATAY: A Streaming Architecture Toolflow for Accelerating YOLO Models on FPGA Devices," 2023 International Conference on Field Programmable Technology (ICFPT), Yokohama, Japan, 2023, pp. 179-187.
   While this publication is not directly related to the core content of this thesis, it originated from a side project in collaboration with Alexander Montgomerie-Corcoran for the AMD Open Hardware Competition 2023[5], where we achieved 1st place in the PhD category.

[5]https://www.openhw.eu/2023-results-gallery

5. **P. Toupas***, Z. Yu*, C.-S. Bouganis and D. Tzovaras, "SMOF: Streaming Modern CNNs on FPGAs with Smart Off-Chip Eviction," 2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Orlando, FL, USA, 2024, pp. 185-196. This publication is relevant to **Chapter 5**.

In the above list of publications, the * symbol denotes equal contribution.

Chapter 2

Background

This chapter provides the necessary background knowledge to contextualize the research contributions presented in this thesis. It covers the key topics important for understanding the techniques and innovations related to FPGA-based acceleration of 3D CNNs presented in subsequent chapters. While this chapter provides a broad overview of the foundational concepts, such as deep learning architectures, hardware accelerators, and the specifics of 3D CNNs, later chapters focus on detailed explanations of the methodologies and techniques applied in this research.

Section 2.1 introduces the fundamentals of deep learning, with a focus on Convolutional Neural Networks (CNNs), including an exploration of both 2D and 3D CNN architectures, a comparison of 2D and 3D CNNs, highlighting the computational and memory challenges transitioning from 2D to 3D space, as well as training and inference procedures. This discussion sets the foundation for understanding the computational challenges of 3D CNNs and highlights the importance of 3D-specific hardware processing blocks and specialised optimisation strategies in making 3D CNNs viable for their efficient deployment on FPGA platforms.

Section 2.2 provides an overview of AI hardware accelerators, outlining the function and capabilities of ASICs, GPUs, and FPGAs. The flexibility and performance advantages of FPGAs are highlighted for customisable, high-efficiency neural network implementations. The discussion encompasses FPGA-specific architectures such as single engine and streaming architectures, with the latter being the method adopted in this thesis. The section also introduces the fpgaConvNet framework, which serves as the methodological basis of this research, investigating the use of synchronous dataflow graphs (SDFGs) and their transformation to efficiently map and optimise CNNs on

FPGA hardware. It also discusses the fundamental concepts that will drive the modelling and design space exploration approaches used later in this thesis.

Finally, Section 2.4 reviews existing 3D CNN FPGA accelerators, presenting the state-of-the-art methodologies and assessing their limitations. By pinpointing these challenges and emphasizing the key areas requiring improvement, this section lays the groundwork for the contributions of this thesis, which aims to bridge these gaps.

2.1 Deep Learning and AI

In contrast to traditional machine learning methods, which necessitate considerable domain expertise and rely on carefully engineered feature extractors, representation learning enables machines to automatically extract useful representations from raw data. Deep learning is such a representation learning technique relying on artificial neural networks (ANNs) with multiple interconnected layers that transform raw inputs into progressively more complex representations. As noted by Yann LeCun, Yoshua Bengio, and Geoffrey Hinton in their seminal Nature publication [27], "Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction.". A key characteristic of deep learning is that the features are not manually designed but learned from data through a general-purpose learning procedure, such as the Stochastic Gradient Descent (SGD) [28].

Inspired by the human brain, neural networks were first introduced by McCullough and Pitts in 1944 [29], who presented the idea of thresholded and weighted connections between neurons. However, these early models lacked a layered structure or a training mechanism. Frank Rosenblatt [30] developed the first trainable neural network, known as the "Perceptron", in 1957, establishing the foundation for modern neural networks by incorporating adjustable weights and thresholds into a single-layer structure.

Deep learning's hierarchical and non-linear way of processing the data has driven remarkable advancements in artificial intelligence (AI), exhibiting outstanding results in tasks such as speech recognition, visual object detection, and applications in domains like drug discovery and genomics. The emergence of it has been driven by the availability of large data sets and powerful computational resources, which enabled the training of deep neural networks at an unprecedented scale. Foundational architectures like Convolutional Neural Networks (CNNs) have been instrumental in handling grid-like data, such as images, while newer models like Transformers have broadened the scope of deep learning to sequential data tasks, like natural language processing. These advances emphasise the

groundbreaking impact and versatility of deep learning in modern AI, with important milestones in its history including:

- AlexNet [11] (2012): Showcased the power of deep learning in image classification by achieving groundbreaking results in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [12]. This success was driven by two key factors: the availability of the large-scale, precisely labelled ImageNet dataset, which provided a rich foundation for training, and the use of GPUs, which dramatically accelerated training by handling parallel computations efficiently. Together, these advancements made it feasible to train a model with approximately 60 million parameters—something that would have been infeasible without the ImageNet dataset and impractically slow using only traditional CPUs. AlexNet's success marked a pivotal moment for Convolutional Neural Networks (CNNs), establishing them as a cornerstone of modern deep learning architectures for image and video related tasks.

- Generative Adversarial Networks (GANs) [31] (2014): GANs enabled the generation of synthetic but realistic data by employing an adversarial training framework, where a generator creates data while a discriminator evaluates its authenticity. Built upon convolutional architectures, GANs demonstrated the versatility of CNNs in generating high-quality images, showcasing their capacity to model complex spatial data. This innovation has revolutionized fields such as image synthesis, style transfer, and creative AI applications, reshaping how AI interacts with creative industries.

- ResNet [9] (2015): Residual Network architecture introduced residual learning through skip connections, which act as shortcuts to bypass layers and improve gradient flow, effectively solving the vanishing gradient problem, a phenomenon where the training signal weakens as it is propagated back through many layers. ResNet extended the scalability of CNNs, making it feasible to train networks with hundreds or even thousands of layers while preserving stability and performance. It demonstrated that deeper architectures could achieve superior accuracy, inspiring subsequent advancements in deep learning design.

- Transformers [10] (2017): They leverage the self-attention mechanism, a technique that allows the model to weigh the importance of different parts of the input data in relation to each other, to capture long-range dependencies and process

data in parallel, enhancing efficiency and scalability by exploiting the available hardware parallelism. This architecture has achieved state-of-the-art results in Natural Language Processing (NLP) and has extended its versatility to domains like computer vision and speech recognition, becoming foundational in modern AI.

2.1.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a cornerstone of modern deep learning, known for being efficient at processing grid-like data, such as images and videos. CNNs, were introduced by Y. LeCun et al. [32] in 1998 with the LeNet-5 architecture for handwritten digit recognition, revolutionizing the design and implementation of Deep Neural Networks (DNNs) on grid-like data.

Prior to the introduction of CNNs, the standard method for training neural networks to categorise images required flattening the image into a one-dimensional vector of pixels and sending it into a Feedforward Neural Network (FNN) to generate the class labels. However, flattening an image not only discards the image's essential spatial information, but also yields a considerably higher number of parameters. This often leads to higher memory demands and computational complexity, especially for larger images, making this method inefficient and resource-intensive.

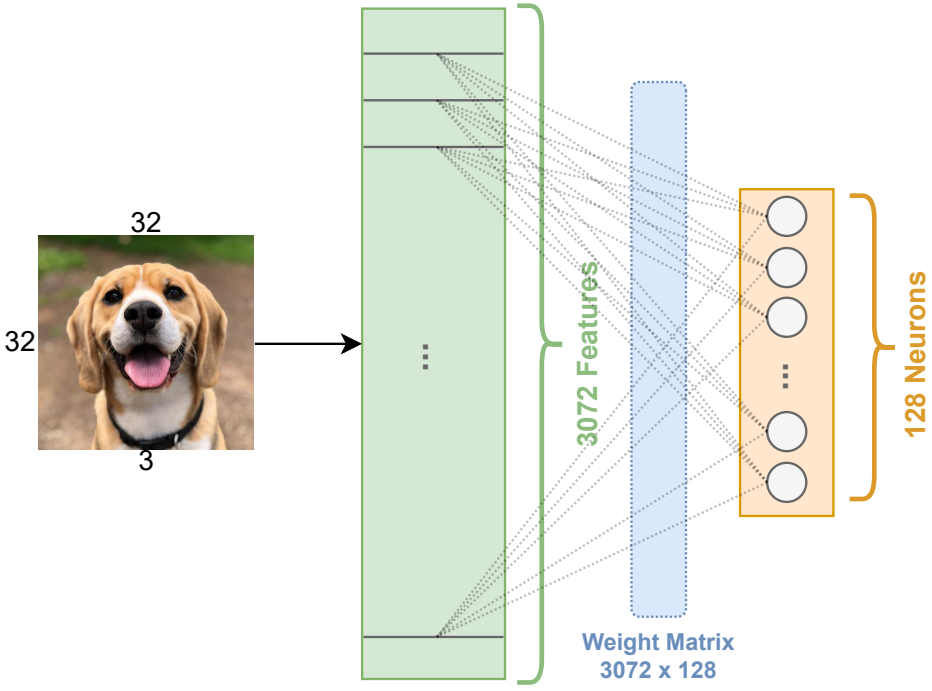


Fig. 2.1 A single-hidden-layer FNN comprising 128 neurons.

CNNs excel at processing grid-like data, such as images, due to their highly efficient processing of the input data. A defining characteristic of CNNs is weight sharing, achieved through the use of convolutional kernels. This method dramatically lowers the number of trainable parameters compared to FNNs, where each neuron in a layer is fully connected to all neurons in the preceding layer. Consider a 32×32 RGB image with three colour channels. An FNN with a single hidden layer of 128 neurons requires $32 \times 32 \times 3 \times 128 = 393,216$ trainable parameters for the first layer alone, as shown in Figure 2.1. On the other hand, a CNN with a convolutional layer of 128 filters (each of size $3 \times 3 \times 3$) requires only $3 \times 3 \times 3 \times 128 = 3456$ parameters. This greatly reduces—over $100\times$ fewer parameters—the memory needs and computational complexity of CNN models.

At the heart of CNNs lie three fundamental layers: convolutional and pooling layers, and nonlinear activation functions, each contributing distinct functionalities to the network's overall capability:

- **Convolutional Layers**: The convolutional layer is the cornerstone of CNNs, responsible for extracting features from input data. It performs convolution operations using a set of learnable filters (or convolutional kernels) that slide over the input, computing dot products at each position. These filters are designed to detect patterns, such as edges, textures, or more complex shapes in deeper layers. The sliding operation, combined with weight sharing, allows the layer to efficiently capture local spatial dependencies while significantly reducing the number of parameters compared to fully connected layers. The output of this layer is a set of feature maps (FMs) that encode the presence and location of learned patterns. A more technical discussion on 2D and 3D convolutional layers will follow in Section 2.1.2.

- **Pooling Layers**: Pooling layers, often applied after convolutional layers, downsample the feature maps. This operation reduces the spatial dimensions of the data, thereby decreasing computational load and mitigating the risk of overfitting. Max pooling, which selects the maximum value in a pooling window, and average pooling, which computes the average, are the two most common pooling techniques.

- **Nonlinear Activation Functions**: Non-linear activation functions, such as ReLU [33], Sigmoid, and Tanh [34], are integral to CNNs and DNNs in general. These functions introduce nonlinearity into the model, enabling the network to learn complex patterns in the data. Without these non-linear operations, the

network would effectively act as a linear function, limiting it to solving only linearly separable problems and preventing it from tackling more complex, non-trivial and non-linear tasks.

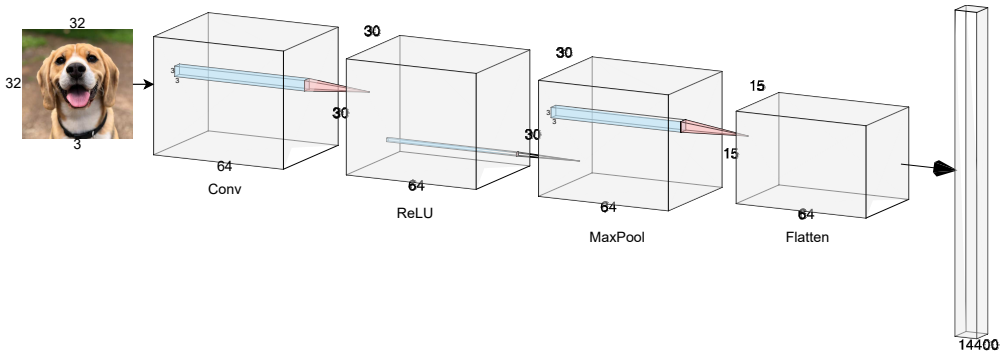


Fig. 2.2 A simple CNN model comprising the three fundamental, Convolution, Pooling, and Nonlinear Activation layers.

Figure 2.2 depicts a very simple CNN model comprising the fundamental layers described above. The input image is processed by a convolutional layer to extract features, then passed through a non-linear activation function (ReLU), and finally downsampled by a pooling layer (MaxPooling) to reduce its spatial dimensions. The final feature map is flattened and passed to a Fully Connected layer (as those used in FNNs).

In addition to decreased number of parameters and computational complexity, CNNs are particularly effective at capturing essential features by focussing on local spatial patterns such as edges and textures rather than depending on the dense connections found in FNNs, which frequently lead to overfitting. Their localised receptive fields allow for hierarchical feature extraction, in which low-level features can be identified in the first layers and more abstract patterns are picked up in deeper layers. These advantages brought CNNs to the forefront of applications such as object detection, image classification, and video recognition, establishing them as a cornerstone of modern AI for processing visual and spatiotemporal data.

2D CNNs

A 2D Convolutional Neural Network (CNN) is designed to process two-dimensional data, such as images, through a layered architecture to extract and learn spatial hierarchies from the input. Despite the fact that 2D CNNs are designed to operate in two dimensions, they are also capable of processing three-dimensional data, such as RGB images. In

this case, each filter covers the whole depth of the input (for example, all three colour channels), allowing for the simultaneous extraction of spatial characteristics across channels. However, the operations themselves—such as convolutions and pooling—are applied strictly along the two spatial dimensions (height and width), preserving the title of "2D" architectures.

A typical 2D CNN architecture is composed of numerous fundamental building components. As previously described, the core of these networks consists of convolutional and pooling layers. Activation functions play an important role in CNN models. These non-linear activation functions, such as ReLU (Rectified Linear Unit), add nonlinearity into the network, allowing it to learn and represent complex patterns. Finally, the extracted features flow through one or more fully connected layers, which map them to output classes or regression targets, concluding the network's decision-making process.

Over the years, several landmark architectures have shaped the evolution of 2D CNNs. These architectures introduced innovations that addressed challenges related to depth, computational efficiency, and scalability. The architectures listed below represent key milestones that emerged following the revolutionary AlexNet [11], which lead to the modern deep learning era.

- GoogLeNet or Inception V1 [35] (2014): Introducing the Inception module, GoogLeNet utilized a multi-branch structure to process features at different scales simultaneously, which are then stacked together to generate the final output, significantly improving computational efficiency without compromising accuracy.

- ResNet [9] (2015): As previously described, ResNet pioneered the concept of residual learning, enabling the training of extremely deep networks by addressing the vanishing gradient problem.

- UNet [36] (2015): Designed specifically for semantic segmentation, UNet utilized an encoder-decoder structure with skip connections, ensuring precise localization of features, which was particularly impactful in medical imaging [37].

- MobileNet [38] (2017): MobileNet focused on mobile and embedded systems, using depthwise separable convolutions to drastically reduce computational requirements while maintaining performance.

3D CNNs

A 3D Convolutional Neural Network (CNN) builds upon the concepts of 2D CNNs to process three-dimensional data, making it particularly suitable for applications requiring the extraction of temporal and volumetric information. In contrast to its 2D equivalents, which function within spatial dimensions (height and width), 3D CNNs incorporate an additional dimension into the convolutional kernels, allowing them to process data along the temporal or depth dimension. Applications on this domain include Human Action Recognition, where 3D CNNs effectively capture temporal dynamics for action classification; medical imaging such as MRI and CT scans, where 3D CNNs help the analysis of spatial structures across slices, improving segmentation; and video analysis, wherein 3D CNNs analyse sequential frames jointly, capturing motion information and contextual relationships crucial for tasks such as video tagging or summarisation. The incorporation of a third dimension increases the computational complexity of the network, making architectural innovations and efficiency optimisations essential for practical deployment.

The architecture of a 3D CNN extends the fundamental operations of 2D CNNs by incorporating a third dimension into convolutional and pooling layers, allowing simultaneous processing and learning of spatiotemporal or depth-related information. This enables 3D CNNs to extract features across width, height, and depth dimensions, offering a unified approach to processing volumetric or video data. This, however, increases computational complexity and memory demands, necessitating meticulous model design and optimizations to achieve efficient implementation.

Several influential architectures have shaped the development of 3D CNNs, addressing challenges related to computational cost, scalability, and accuracy in spatiotemporal data analysis:

- C3D (Convolutional 3D Network) [39] (2015): One of the first 3D CNNs, C3D demonstrated the potential of 3D convolutions for spatiotemporal feature learning. It applied 3D kernels to video data, effectively capturing both spatial and temporal dependencies, making it a benchmark in video analysis, particularly in the task of Human Action Recognition (HAR).

- Temporal Segment Networks (TSN) [40] (2016): TSN introduced a novel segment-based sampling and aggregation approach. By dividing the videos into segments and sampling from these segments it could efficiently capture long-term temporal dependencies. Utilising its aggregation mechanism to combine the information

from these sampled segments TSN could form a comprehensive understanding of the action being performed.

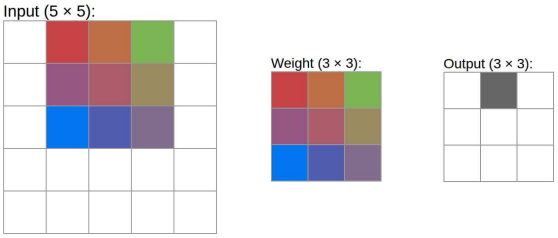
- Inflated 3D ConvNet (I3D) [41] (2017): I3D expanded 2D CNN architectures, like Inception V1 [35], by inflating filters and pooling kernels into three dimensions, effectively "transferring" knowledge from pre-trained 2D CNNs to 3D CNNs. Additionally, it employed a two-stream architecture style, processing both RGB frames and optical flow to capture appearance and motion, with predictions averaged during testing to enhance accuracy.

- SlowFast Networks [42] (2019): These networks used a two-stream architecture style to process video data at different temporal resolutions. The "Slow" pathway captured spatial semantics at a lower frame rate, while the "Fast" pathway focused on dynamic temporal changes, achieving a balance between accuracy and efficiency.

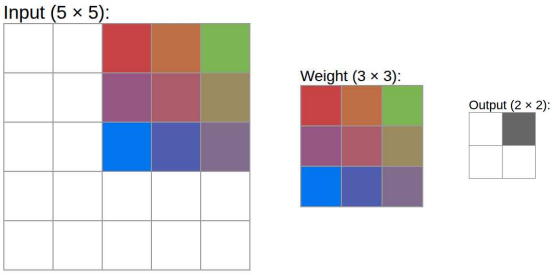
- X3D [6] (2020): X3D explored the idea of scaling 3D CNNs in multiple dimensions—such as width, depth, and resolution—tailoring the architecture for video analysis while optimizing for computational efficiency. Its scalability and adaptability made it a versatile choice for various spatiotemporal tasks.

2.1.2 2D vs 3D Convolution

Convolution is the fundamental operation in Convolutional Neural Networks (CNNs), and it is essential for feature extraction and representation learning. Convolutional layers work by sliding a kernel (or filter) across the input data, executing dot-product computations between the kernel and overlapping regions of the input. The core parameters of the convolution operation include kernel size, stride, padding, and dilation, as detailed below, while Figure 2.3 depicts the effect of different parameter configurations on a 2D convolution.



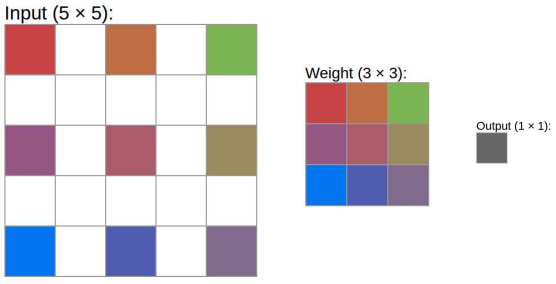
(a) Convolution operation with the following configuration:
Input size (5 × 5), Kernel size (3 × 3), Stride (1), Padding (0), Dilation (1).



(b) Convolution operation with the following configuration:
Input size (5 × 5), Kernel size (3 × 3), Stride (2), Padding (0), Dilation (1).



(c) Convolution operation with the following configuration:
Input size (5 × 5), Kernel size (3 × 3), Stride (1), Padding (1), Dilation (1).



(d) Convolution operation with the following configuration:
Input size (5 × 5), Kernel size (3 × 3), Stride (1), Padding (0), Dilation (2).

Fig. 2.3 Visual representation of the Convolution in various configurations of the main parameters: Kernel Size, Stride, Padding, and Dilation.

- **Kernel Size**: It determines the field of view or receptive area of the convolution. In 2D CNNs, kernel sizes are typically 3×3 or 5×5, while in 3D CNNs, the kernels are extended to $3 \times 3 \times 3$, reflecting the additional spatial-temporal dimension.
- **Stride**: It determines the kernel's step size while it traverses the input. A stride of one preserves the spatial dimensions, while strides greater than one result in downsampling of the spatial dimensions, similar to pooling operation.
- **Padding**: It specifies how the input's edges are handled. Padding can either keep the input size same (e.g., "same" or half padding) or reduce it (e.g., "valid" or no padding), changing the output's spatial dimensions.
- **Dilation**: Controls the spacing between kernel values, allowing for larger receptive fields while keeping the number of parameters unchanged. For example, a 3×3 kernel with a dilation of two will have the same field of view as a 5×5 kernel while only needing 9 parameters.
- **Input and Output Channels**: A convolutional layer receives input data with a fixed number of channels C (for example, three for RGB images) and generates outputs with a configurable number of output channels, often referred to as filters F. Each filter corresponds to a distinct set of weights used to extract specific features from the input data, resulting in a unique output channel in the output feature map.

Both 2D and 3D CNNs employ various convolution types to optimize performance, enhance feature extraction, or reduce computational cost. Key convolution types include:

- **Full Convolution**: This operation computes each output channel by applying a kernel to all input channels and aggregating their results. The kernel processes the input by sliding over it in two dimensions (height and width), performing element-wise multiplications, and summing the results to generate a single pixel per window. This process is repeated throughout the input, yielding a feature map that encodes spatial correlations. In 3D CNNs, this concept is applied to volumetric data, with the kernel operating in three dimensions—height, width, and depth (or time)—to capture both spatial and temporal relationships. The technical details of how these processes vary with input and output configurations will be discussed later on in the section.

- **Grouped Convolution**: Initially introduced in AlexNet [11] (2012), grouped convolution splits input channels into separate groups, applying convolutions independently within each group reducing the number of parameters and computations compared to full convolution.

- **Pointwise Convolution**: This type of convolution initially proposed by M. Lin et al. [43] (2014) employs a 1×1 kernel in 2D or a $1 \times 1 \times 1$ kernel in 3D to operate on relevant spatial or spatial-temporal positions across the channel depth. It is typically used for dimensionality reduction by modifying the number of output channels and for mixing features across channels while maintaining spatial resolution.

- **Depthwise Convolution**: Introduced by F. Chollet [44] (2017), depthwise convolution is a variation of standard convolution where each input channel is processed independently. Instead of applying a single filter to all input channels combined (as in a standard convolution), depthwise convolution assigns a separate filter to each input channel, performing convolution operations independently for each channel. This approach reduces computational complexity while preserving spatial information within each channel. When paired with pointwise convolution, which integrates features across channels, it forms Depthwise Separable Convolutions, which are the fundamental block of efficient architectures like MobileNet [38] or X3D [6].

- **Dilated/Atrous Convolution**: Initially proposed by L-C. Chen et al. [45] (2017), dilated convolution expands the kernel's receptive field by incorporating a dilation rate, that "inflates" the kernel by inserting spaces between its elements, without increasing the number of parameters. This approach provides a broader field of view while maintaining the same computational cost, making it particularly effective for tasks like real-time segmentation where efficiency and a wide receptive field are critical.

Kumar et al. [46] provide a comprehensive overview of the various convolution operations with detailed, animated, and interactive set of diagrams to illustrate the distinctions between every variant.

Throughout the main chapters of this thesis, the types of convolution used in the chosen CNNs for mapping and optimisation on reconfigurable hardware will be examined further.

Key Differences in 2D and 3D Operations

While 2D CNNs are primarily applied to image-based tasks, where inputs are represented as 2D spatial grids, 3D CNNs operate on volumetric data, such as videos or 3D medical imaging, accounting for the additional temporal or depth dimension. This extension introduces challenges related to computational complexity, memory requirements, and efficient hardware implementation.

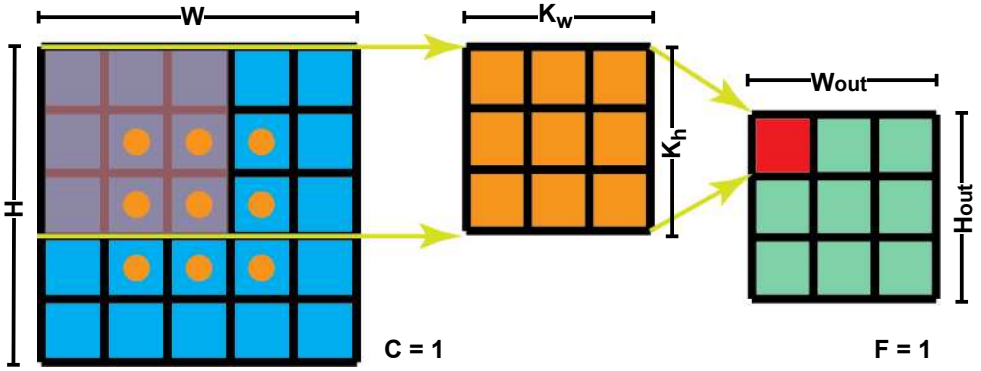


Fig. 2.4 Operation of 2D convolution on single channel input feature map

To better understand the key differences between 2D and 3D convolutions, it is essential to examine their operations, inputs, outputs, and kernels.

Given an input feature map (S^{in}) of shape $\{H \times W \times C\}$ where H, W, C are the input height, width, and channels and a set of filters F of shape $C \times K_h \times K_w$ where K_h, K_w are the kernel's height and width, a 2D convolution operation results in an output feature map (S^{out}) of shape $H_{out} \times W_{out} \times F$ where H_{out} and W_{out} are the output height and width, and are calculated as follows.

$$
\begin{aligned}
H_{out} &= \left\lfloor \frac{H + (P_{h_s} + P_{h_e}) - DR_h \times (\mathrm{K_h} - 1) - 1}{J_h} + 1 \right\rfloor \\
W_{out} &= \left\lfloor \frac{W + (P_{w_s} + P_{w_e}) - DR_w \times (\mathrm{K_w} - 1) - 1}{J_w} + 1 \right\rfloor
\end{aligned} \tag{2.1}
$$

Here P_{h_s}, P_{w_s} represent the padding applied to the top and left sides of the input image, while P_{h_e}, P_{w_e} denote the padding added to the bottom and right sides. The dilation rates in the height and width dimensions are specified by DR_h, DR_w and the strides along the height and width of the image are given by J_h, J_w.

In the 2D version, the kernel moves across the height and width dimensions of the feature map, computing the output pixel value for each window of the input, as illustrated

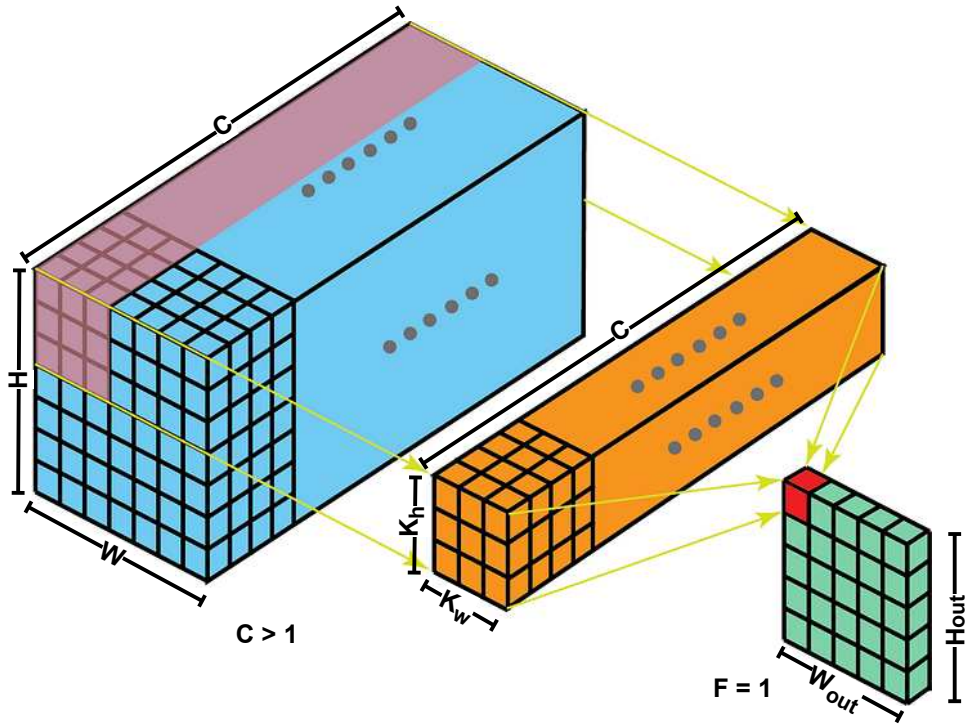


Fig. 2.5 Operation of 2D convolution on multiple channel input feature map

in Figure 2.4. Mathematically, for a 2D convolution with a stride and dilation of 1, each pixel of the output feature map S^{out} at position (f, h, w) is computed as follows:

$$S^{out}(f,h,w) = \sum_{c=1}^{C} \sum_{k_h=1}^{K_h} \sum_{k_w=1}^{K_w} S^{in}(c, h+k_h, w+k_w) \cdot W(f,c,k_h,k_w) \tag{2.2}$$

In convolution with a single input channel (Figure 2.4), the kernel performs convolution on the input feature map to generate an output feature map. When there are multiple input channels (Figure 2.5), each channel has its own kernel, and convolution is done independently for each channel. The final output feature map is produced by accumulating the results from all input channels (summing them element-wise). This accumulation enables convolution to integrate information from all input channels.

The computational complexity of the 2D convolution, in terms of MAC operations, is equal to $C \times K_h \times K_w \times H_{out} \times W_{out} \times F$, while the number of learnable parameters are $C \times K_h \times K_w \times F$.

While 2D CNNs are tailored for image data, 3D CNNs extend the concept to volumetric data or spatiotemporal data, such as video sequences or medical imaging volumes. This extension involves using 3D kernels, which traverse both spatial dimensions and an

additional depth (or temporal) dimension, enabling the network to learn both spatial and temporal correlations.

In the case of 3D convolutions with an input feature map (S^{in}) of shape $\{D \times H \times W \times C\}$, where D, H, W, C are the input depth, height, width, and channels, and a set of filters F of shape $C \times K_d \times K_h \times K_w$ where K_d, K_h, K_w are the kernel's depth, height, and width, a 3D convolution results in an output feature map (S^{out}) of shape $H_{out} \times W_{out} \times D_{out} \times F$ where D_{out}, H_{out}, W_{out} are the output depth, height, and width, and are calculated as follows.

$$\begin{aligned} D_{out} &= \left\lfloor \frac{D + (P_{d_s} + P_{d_e}) - DR_d \times (\mathrm{K_d} - 1) - 1}{J_d} + 1 \right\rfloor \\ H_{out} &= \left\lfloor \frac{H + (P_{h_s} + P_{h_e}) - DR_h \times (\mathrm{K_h} - 1) - 1}{J_h} + 1 \right\rfloor \\ W_{out} &= \left\lfloor \frac{W + (P_{w_s} + P_{w_e}) - DR_w \times (\mathrm{K_w} - 1) - 1}{J_w} + 1 \right\rfloor \end{aligned} \tag{2.3}$$

Here P_{d_s} represents the padding applied to front side of the input volume, while P_{d_e} denotes the padding added to the back side of it. The dilation rate in the depth dimension is specified by DR_d and the stride along the depth of the volume is given by J_d.

In the 3D case, the kernel is extended to include the additional depth dimension. This expansion also applies to the input and output feature maps, in which the kernel traverses not only the height and width dimensions, but also the depth dimension to account for the volume's depth. The 3D convolution operation is illustrated in Figure 2.6, highlighting one fundamental distinction between 2D and 3D convolutions: the incorporation of the depth dimension in the kernel's traversal. Mathematically, for 3D convolution with stride and dilation equal to 1, each pixel of the output feature map S^{out} at position (f, d, h, w) is computed as follows:

$$S^{out}(f, d, h, w) = \sum_{c=1}^{C} \sum_{k_d=1}^{K_d} \sum_{k_h=1}^{K_h} \sum_{k_w=1}^{K_w} S^{in}(c, d + k_d, h + k_h, w + k_w) \cdot W(f, c, k_d, k_h, k_w) \tag{2.4}$$

The computational complexity of the 3D convolution, in terms of MAC operations, is equal to $C \times K_d \times K_h \times K_w \times D_{out} \times H_{out} \times W_{out} \times F$, while the number of learnable parameters are $C \times K_d \times K_h \times K_w \times F$.

When comparing the computational cost of 2D and 3D convolutions, the additional depth dimension in 3D convolutions increases the overall complexity. The total compu-

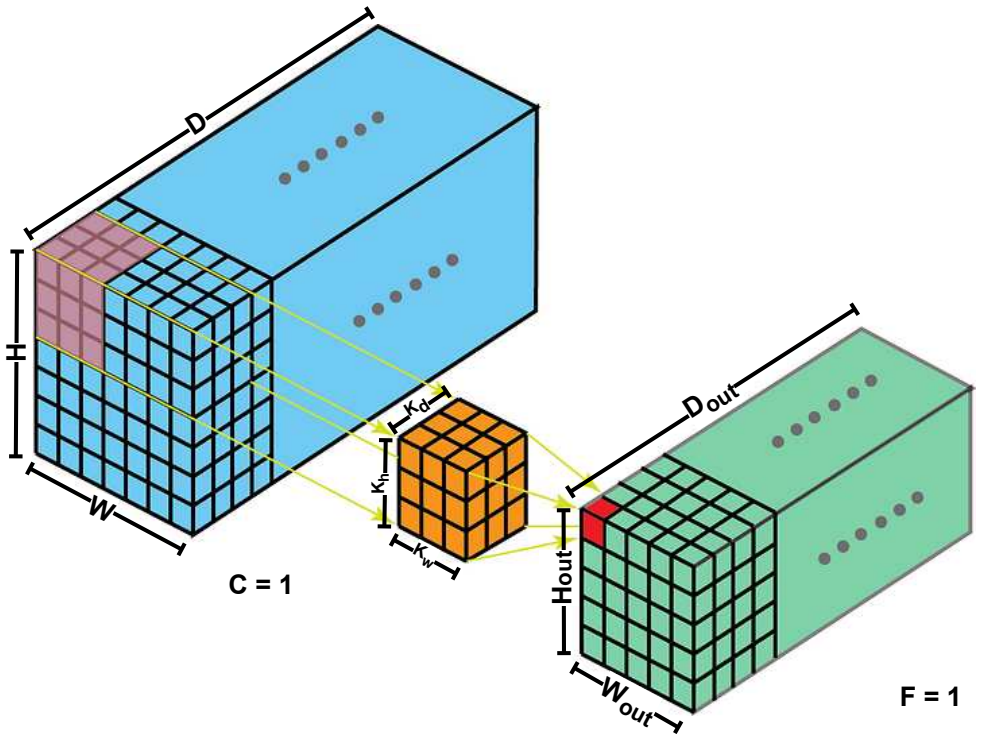


Fig. 2.6 Operation of 3D convolution on single channel input feature map

tations grow proportionally to $K_d \times D_{out}$, where K_d is the kernel size along the depth dimension, and D_{out} is the depth of the output feature map. This added dimension also escalates the memory requirements, as the filters extend into three dimensions, increasing the number of parameters by a factor of K_d for each filter.

A detailed comparison of 2D and 3D variants of four modern CNN models, ResNet [9], ShuffleNetV2 [47], SqueezeNet [48], and UNet [36], is presented in Table 2.1. Furthermore Figures 2.7 to 2.10 present more detailed, per (convolutional) layer information for the ResNet, ShuffleNetV2, SqueezeNet, and UNet models, respectively. These figures illustrate the differences in Multiply and Accumulate (MAC) operations, intermediate feature map memory usage in MBs, and the number of parameters on a per-layer basis, focusing exclusively on the convolutional layers of each model.

By further analysing the data of Table 2.1 we observe that 3D CNNs exhibit a 6.28× higher volume of intermediate activations, measured in MBs, owing to the extra depth dimension. This additional dimension greatly expands the data processed at each layer, leading to a substantial increase in memory. Furthermore, 3D models demand an average of 8.96× more Multiply and Accumulate (MAC) operations compared to their 2D counterparts, reflecting the greater computational complexity introduced by

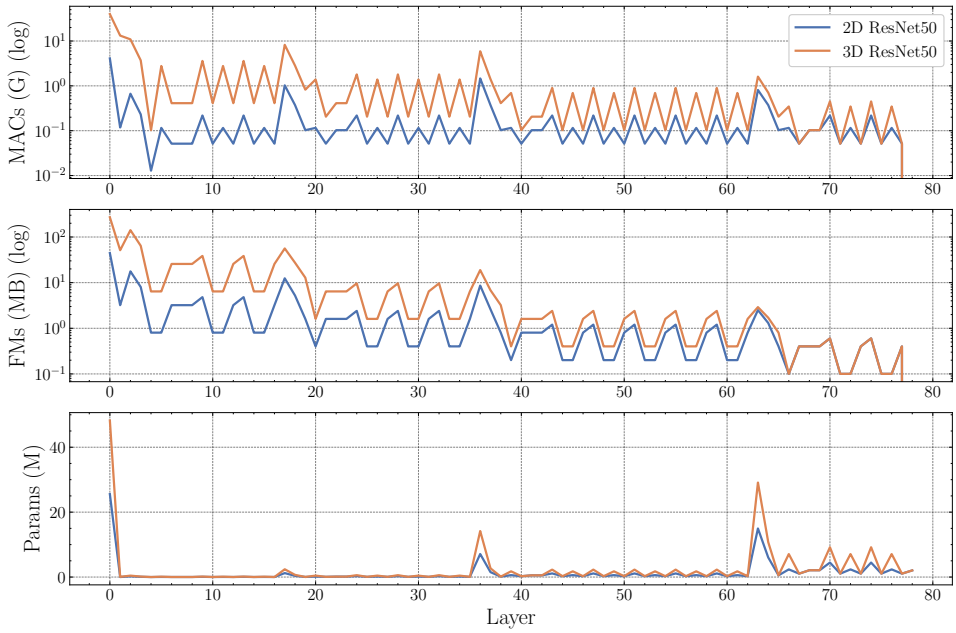


Fig. 2.7 Comparing ResNet Model Characteristics and Statistics between 2D and 3D Versions

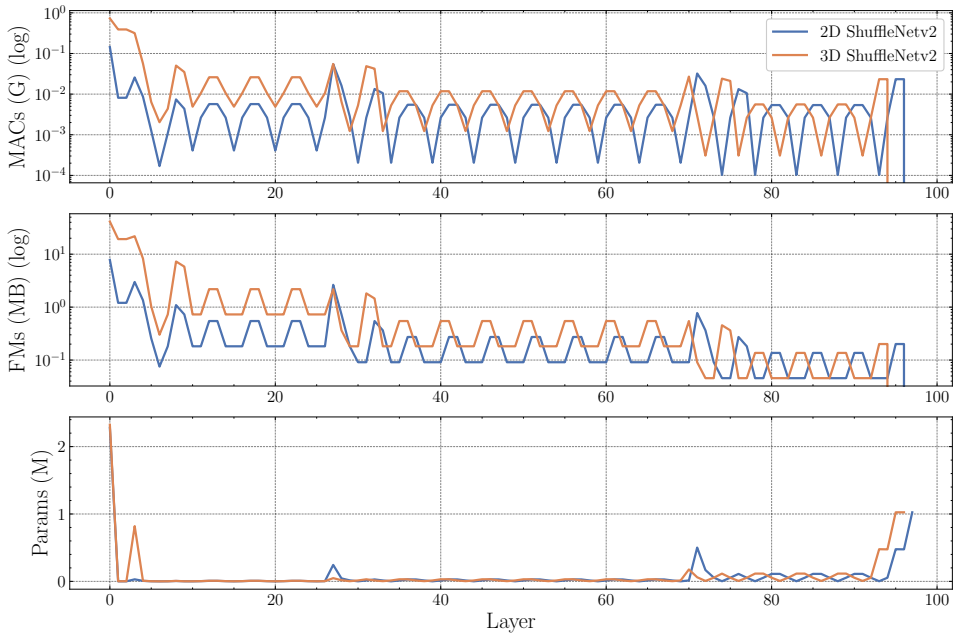


Fig. 2.8 Comparing ShuffleNetv2 Model Characteristics and Statistics between 2D and 3D Versions

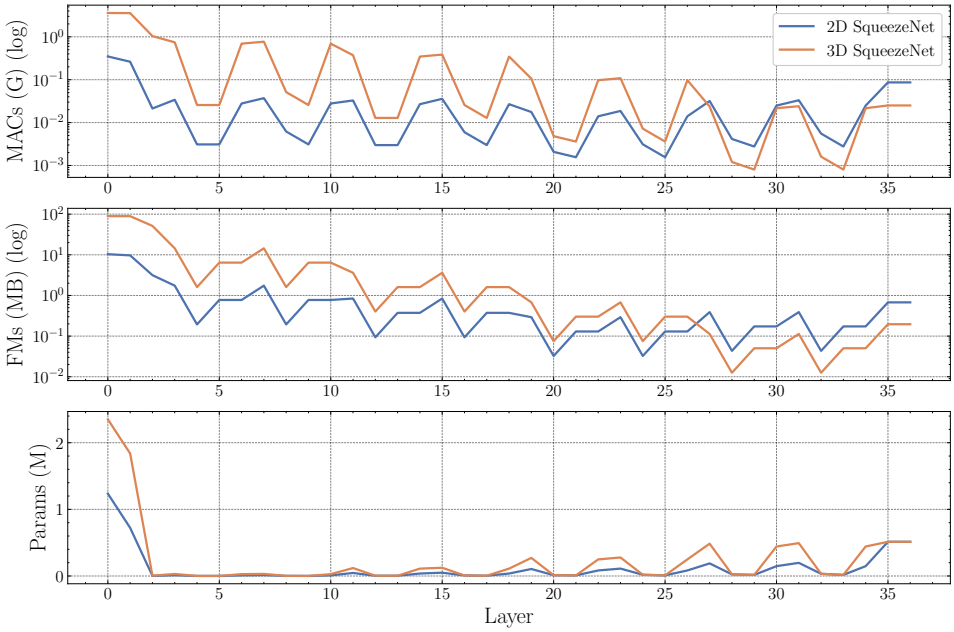


Fig. 2.9 Comparing SqueezeNet Model Characteristics and Statistics between 2D and 3D Versions

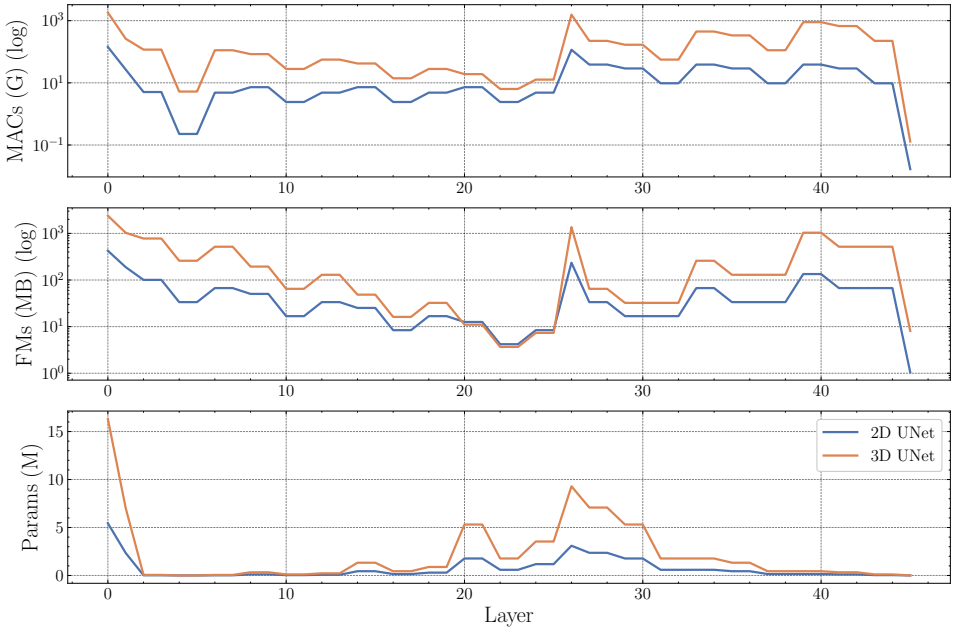


Fig. 2.10 Comparing UNet Model Characteristics and Statistics between 2D and 3D Versions

Table 2.1 Characteristics and statistics on modern CNN architectures between their 2D and 3D versions

	Input Dims	MACs (G)	Params (M)	Activations (MB)
		2D		
ResNet50	224x224	4.11	25.55	44.46
ShuffleNetV2	224x224	0.15	2.28	7.81
SqueezeNet	224x224	0.35	1.23	10.36
UNet	512x512	143.59	5.44	424.67
		3D		
ResNet50	16x224x224	39.98	48.25	270.65
ShuffleNetV2	16x224x224	0.75	2.32	41.28
SqueezeNet	16x224x224	3.65	2.35	89.28
UNet	116x132x132	1830.54	16.32	2396.01

the volumetric nature of 3D convolutions. Additionally, the number of parameters in 3D models is 1.82× higher on average, as the convolutional kernels extend into the depth dimension, resulting in more complex and resource-intensive filters.

These results highlight the significantly greater computational and memory demands of 3D CNNs compared to 2D CNNs, as evidenced by the stark differences in parameter count, activation volumes, and MAC requirements. The increased resource intensity of 3D models underscores the need for 3D-specific optimizations and hardware accelerators to make their use feasible in real-world applications.

2.1.3 CNN Training and Inference

Convolutional Neural Networks (CNNs) operate through two distinct phases: training and inference. These phases differ significantly in terms of objectives, computational requirements, and operations performed, yet they collectively enable CNNs to learn and utilize meaningful representations of data.

Training: Learning Representations: During the training phase, the objective is to find a function that maps input data x to the desired outputs y. This approach is a form of supervised learning, as the model is trained on a dataset where each input is paired with a known, correct output (i.e., labelled data). This process begins with forward propagation, where x passes through the network layers, undergoing operations like convolution, activation, and pooling, to produce a predicted output $\hat{y}$. The predicted $\hat{y}$ is compared to the ground truth y using a loss function, $J(\theta)$ which calculates a scalar cost, where θ represents the network's learnable parameters (weights and biases). Common loss functions include Mean Squared Error for regression and Cross-Entropy Loss for classification. Backpropagation, introduced by Rumelhart et al. [34] in 1986,

is an algorithm used to compute the gradients of the loss function $J(\theta)$ with respect to each parameter in the network. It does that by propagating the error backward through the network after a forward pass. These gradients are essential for understanding how much each weight contributes to the error, guiding the optimization process. The calculated gradients are then utilised by optimization algorithms such as Stochastic Gradient Descent (SGD) [28] or Adam [49] to iteratively update the parameters, θ, of the model. These optimisation algorithms are designed to minimize the loss function by adjusting the model's weights in the direction of the steepest descent, as indicated by the computed gradients. This iterative cycle—forward propagation, loss computation, backpropagation, and parameters updates—continues until convergence criteria are met.

Inference: Utilizing Learned Representations: Once a model is trained, meaning its parameters θ have been optimized through the training process, it can be used for inference. During inference, the trained network is deployed to process unseen input data, leveraging the parameters θ learned during training. The primary objective of inference is to deliver accurate and reliable predictions $\hat{y}$. Unlike training, which involves multiple steps such as forward propagation, loss computation, backpropagation, and parameters updates, inference focuses solely on forward propagation. This significantly reduces the computational cost and memory requirements.

The computational cost, memory usage, and hardware requirements of these phases vary significantly. Training is computationally intensive, often demanding high-performance resources like GPUs or distributed systems to handle the extensive data processing and iterative updates involved in optimizing model parameters. In contrast, inference focuses on real-time or near-real-time applications, prioritizing low latency and efficient resource usage to operate within the constraints of deployment environments, such as edge devices or embedded systems. This thesis concentrates on the inference process of CNNs. By leveraging reconfigurable hardware, specifically Field-Programmable Gate Arrays (FPGAs), this work addresses the need for efficient, scalable inference solutions tailored to the requirements of modern AI applications. The training phase is discussed only to provide foundational understanding and context for the inference-specific contributions of this research.

2.2 AI Hardware Accelerators

Deep learning has revolutionized many different areas, including image and video analysis, natural language processing, and medical diagnostics. Nonetheless, its transformative

potential comes at a cost: the significantly high computational demands required to deploy these models. These demands emerge from the complexity and scale of deep learning architectures. For example, Convolutional Neural Networks (CNNs)—a cornerstone of Computer Vision applications—are particularly demanding due to their large workloads, incorporating millions of parameters. This necessitates highly efficient and optimized ways to perform operations such as Convolution and General Matrix Multiplication (GEMM).

To address these challenges, hardware acceleration has become essential. CNN workloads are fundamentally parallelisable since they depend on operations like Convolution and General Matrix Multiplication (GEMM), which can be performed concurrently across multiple processing units. By leveraging this parallelism, hardware accelerators transform the high computational demands into feasible solutions that can provide real-time or near-real-time performance. The necessity for hardware accelerators is particularly evident when considering computationally demanding models such as Transformers or 3D CNNs.

Among the platforms enabling this transformation are Application-Specific Integrated Circuits (ASICs), General Purpose Graphics Processing Units (GPUs), and Field-Programmable Gate Arrays (FPGAs). Each of these technologies provides unique advantages tailored to the needs of deep learning applications. **ASICs** provide exceptional efficiency and performance for designated tasks, although they lack flexibility and are expensive to develop and fabricate. **GPUs**, characterised by their large amount of processing cores and high memory bandwidth, have emerged as the preferred choice for deep learning training. They achieve a balance between performance and programmability, although they tend to be power-hungry, making them less suitable for applications targeting edge devices or embedded systems. **FPGAs**, the focus of this research, provide a compelling middle ground, delivering hardware-level parallelism and customisation while maintaining a high degree of flexibility. Their reconfigurable nature makes them well-suited for accommodating a variety of deep learning models and workloads, especially those with unique characteristics and demands.

The rest of this section will explore the architectural characteristics and capabilities of the three platforms—ASICs, GPUs, and FPGAs—shedding light on their contributions to the advancement of deep learning.

2.2.1 Application-Specific Integrated Circuits (ASICs)

Application-Specific Integrated Circuits (ASICs) are specialised hardware accelerators tailored for particular algorithms or computational tasks. Their co-designed hardware and software architecture offers exceptional performance and efficiency, evidenced by devices such as Google's Tensor Processing Unit (TPU) [13] and Groq's Tensor Streaming Processors [14]. This optimisation results in minimum overhead and a compact silicon footprint, making ASICs suitable for large-scale industrial and commercial applications. Nonetheless, ASIC development is expensive and time-consuming, necessitating substantial expertise and resources. In contrast to FPGAs or GPUs, ASICs offer no flexibility; once manufactured, their architecture cannot adapt to changing algorithms or model-specific enhancements, hence constraining their applicability in dynamic or exploratory research settings. Despite these limitations, ASICs exhibit great efficiency for stable, fixed-function applications, making them the preferred option for production-scale implementations when performance and energy efficiency are critical.

2.2.2 Graphics Processing Units (GPUs)

Graphics Processing Units (GPUs) are essential for deep learning due to their capacity for extensive parallel computations. Utilising Single-Instruction-Multiple-Data (SIMD) parallelism, GPUs provide exceptional performance in training deep neural networks (DNNs) owing to their parallel and pipelined architecture. Their rich ecosystem, featuring APIs such as CUDA[1], OpenCL[2], and SDKs like TensorRT [21, 22], facilitates fine-grained optimization and seamless integration with the most popular deep learning frameworks like PyTorch [50], TensorFlow [51], ONNX[3], and Caffe [52].

Although GPUs provide outstanding performance, especially on DNN training tasks, they are characterised by significant high power consumption, space requirements, and heating problems making them not an ideal candidate platform for edge applications. Embedded GPUs, like NVIDIA Jetson devices[4], mitigate these constraints by focussing on lightweight network architectures utilising techniques such as quantisation and pruning, even though this results in decreased performance compared to their high-end counterparts.

[1] https://developer.nvidia.com/cuda-toolkit
[2] https://www.khronos.org/opencl/
[3] https://onnx.ai/
[4] https://developer.nvidia.com/embedded-computing

In essence, GPUs serve as a powerful and versatile platform for accelerating deep learning, especially for training tasks and scenarios requiring high throughput. Despite their advantages, GPUs depend significantly on the regularity and uniformity of workloads to attain optimal performance. This dependence makes them less suitable than ASICs or FPGAs for applications necessitating optimised latency reductions or task-specific acceleration. Moreover, the high power consumption and maintenance costs of traditional GPUs pose challenges for deployment in resource-constrained environments, where energy efficiency, and cost-effectiveness are critical considerations.

2.2.3 Field-Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays (FPGAs) hold a unique position in the hardware acceleration landscape, providing a middle ground between the high efficiency of ASICs and the flexibility of GPUs. Compared to GPUs, FPGAs offer greater hardware-level customisation, facilitating optimised implementations tailored to specific workloads, making them particularly effective for inference applications where energy efficiency and latency are essential. Compared to ASICs, which necessitate extensive development cycles and substantial non-recurring engineering costs, FPGAs offer a pre-fabricated, readily available option that can be programmed to implement end-to-end solutions within weeks. By bypassing processes like physical design, layout, and fabrication, FPGAs dramatically reduce time-to-market and development complexity. Their reconfigurable nature and capacity for rapid prototyping enable FPGAs to adapt to evolving requirements, offering a degree of flexibility unmatched by ASICs or GPUs.

FPGAs' distinctive combination of flexibility, efficiency, and capacity to quickly adapt to evolving workloads makes them notably well-suited for deep learning applications, particularly in light of the growing need for hardware architectures that balance high performance, power efficiency, and customisation. By bridging the gap between the programmability of power-hungry GPUs and the efficiency of fixed-function ASICs, FPGAs offer an attractive alternative for AI-specific applications. Several studies [53–56] have highlighted the strengths of FPGA-based accelerators, providing detailed comparisons of device options and methodologies for optimizing deep neural network architectures.

Venieris et al. [57] categorised FPGA-based AI accelerator designs into two main architecture styles: single engine and streaming architectures. These architectural paradigms illustrate the adaptability of FPGAs, demonstrating their ability to accommodate the diverse and evolving requirements of deep learning algorithms. The combination of

reconfigurability, efficiency, and adaptability makes FPGAs a suitable platform for the advancement of AI acceleration.

FPGA Resource Primitives

The reconfigurable fabric of modern FPGAs is heterogeneous, comprising several types of specialised hardware blocks. Understanding these fundamental resource primitives is crucial for designing and optimising CNN accelerators, as the efficiency of an implementation is determined by how well it utilizes these resources. The primary resource types relevant to this thesis are:

- **Logic Blocks**: The most fundamental components of an FPGA are the logic blocks, which consist of Look-Up Tables (LUTs) and Flip-Flops (FFs). LUTs are small memory blocks that can be programmed to implement any Boolean logic function of a few inputs (typically 4-6). FFs are used to store state and register signals, enabling the creation of sequential logic and pipelines. LUTs can also be configured as small, fast distributed RAM blocks, often referred to as LUTRAMs, which are ideal for implementing small buffers or FIFOs directly within the logic fabric.

- **Digital Signal Processing (DSP) Slices**: Modern FPGAs integrate dedicated DSP slices, which are hardened arithmetic logic units optimized for high-performance signal processing tasks. For CNN acceleration, their most critical function is performing Multiply-Accumulate (MAC) operations efficiently. A single DSP slice can execute an integer multiplication and subsequent accumulation in a single clock cycle with high performance and power efficiency. This makes them indispensable for implementing the computationally intensive convolutional and fully connected layers, which are fundamentally composed of a vast number of MAC operations.

- **On-Chip Memory Blocks**: FPGAs feature dedicated on-chip memory blocks to store data locally, enabling high-bandwidth access and reducing reliance on slower off-chip memory. These come in two main forms:

  - **Block RAM (BRAM)**: These are relatively small, dual-port static RAM blocks distributed throughout the FPGA fabric. Their dual-port nature allows simultaneous read and write operations, making them highly effective for

storing model weights, biases, and intermediate feature maps that need to be accessed frequently within the dataflow.

- **UltraRAM (URAM)**: Available on newer, high-end FPGA families, URAMs are much larger, single-port memory blocks that offer a higher storage density than BRAMs. While they lack the dual-port capability of BRAMs, their large capacity makes them invaluable for storing the substantial parameter sets or large activation volumes characteristic of modern, deep CNNs, especially 3D models.

The strategic allocation and management of these diverse resources are central to the design of high-performance, efficient FPGA-based accelerators, a theme that is explored in detail in the subsequent chapters of this thesis.

FPGA Design Flow

Translating a hardware design into a bitstream, that can be loaded onto an FPGA is a complex, multi-stage process known as the design flow. This process is computationally intensive and explains the long compilation times associated with FPGA design development. The primary stages include:

- **Synthesis**: The design, typically described in a Hardware Description Language (HDL) like Verilog or VHDL (often generated from a High-Level Synthesis tool), is translated into a netlist. This netlist represents the design as a collection of the FPGA's primitive resources (LUTs, FFs, DSPs, BRAMs) and the logical connections between them.

- **Placement**: In this stage, each primitive element from the netlist is assigned a specific physical location on the FPGA's silicon die. This is a highly complex combinatorial optimization problem, as the placement directly impacts the feasibility and performance of the subsequent routing stage.

- **Routing**: After placement, the programmable interconnects of the FPGA are configured to create physical wire connections between the placed components, realizing the logical connections defined in the netlist. This is another computationally demanding optimization task, as the router must manage limited wiring resources to meet timing constraints.

- **Bitstream Generation**: Finally, the placed and routed design is converted into a binary bitstream file that contains the full configuration data for every programmable element on the FPGA.

The placement and routing stages are particularly time-consuming, often taking several hours or even days for large and complex designs. This long compilation cycle makes it impractical to iterate through many design variations by running the full flow for each one. This reality strongly motivates the approach taken in this thesis: using accurate performance and resource models to conduct an efficient Design Space Exploration (DSE). By modelling the design's behaviour, we can rapidly evaluate thousands of potential configurations and identify promising candidates before committing to the lengthy process of generating a final bitstream.

Single Engine Architectures

Single engine architectures are designed to optimise General Matrix Multiplication (GEMM), a core operation in AI workloads, applicable in fully connected and convolutional layers, which can be represented as matrix multiplication [58–62]. The core of single engine architectures usually comprises a systolic array, which is a grid-like array of processing elements (PEs), that enables parallel computations as depicted in Figure 2.11. One can think of the entire systolic array as a single, powerful, and re-taskable math engine. Each individual PE performs a fundamental task—typically a single Multiply-Accumulate (MAC) operation. The entire array of PEs must work together to compute the workload of a full CNN layer (e.g., a convolution). Once a layer is complete, the engine is immediately re-tasked via software instructions to process the next layer, reusing the exact same hardware. This approach is highly flexible, as the same engine can be instructed to compute any layer type, but it processes the network one layer at a time. Data flows through the array, with partial products accumulating as they traverse the PEs. Input data and weights are typically stored in on-chip buffers, reducing the latency associated with off-chip memory accesses and improving overall computational throughput.

One of the defining characteristics of single engine architectures is their ability to generalize across a wide range of convolutional neural network (CNN) models. This adaptability is made possible through flexible scheduling mechanisms and software-controlled operations. Each CNN corresponds to a unique sequence of hardware-executable microinstructions, allowing the same underlying hardware architecture to be configured

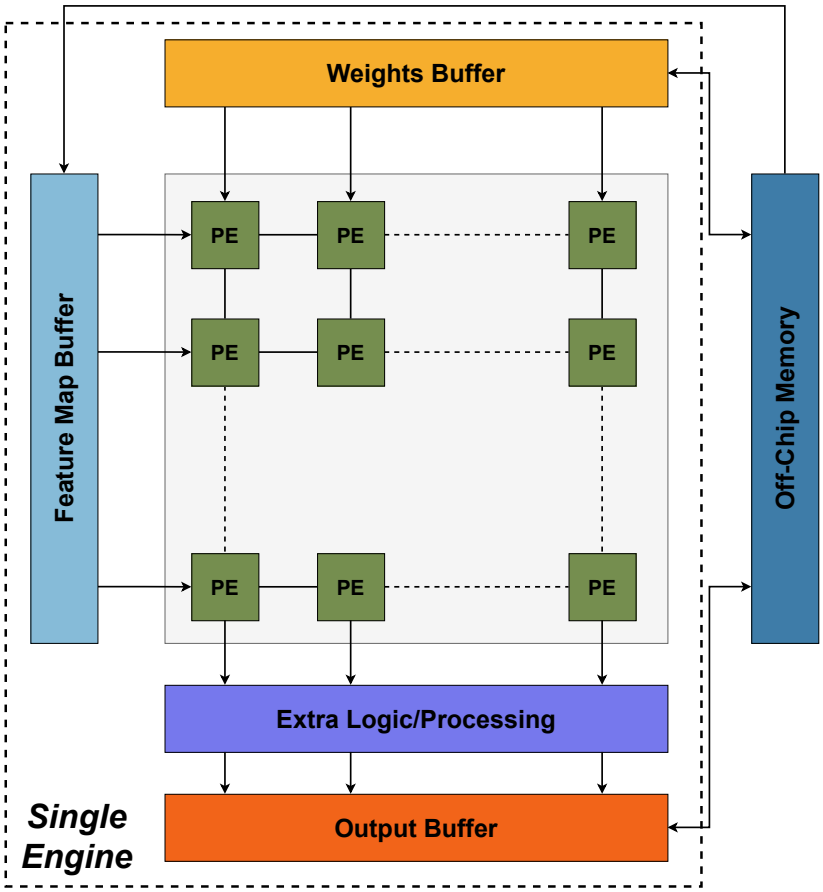


Fig. 2.11 Single engine architecture, in the form of a systolic array, for FPGA-based DNN acceleration, featuring a grid of Processing Elements (PEs), on-chip buffers for weights, feature maps, and outputs, communicating with the off-chip memory for data loading and offloading.

and adapted to different CNN workloads. Domain-specific instruction sets and dedicated compilers have also been developed to streamline hardware control and facilitate efficient scheduling [63, 64].

Despite these advantages, single engine architectures still have some limitations. They are designed as a one-size-fits-all solution, prioritising adaptability over task-specific customisation. This flexibility often leads to inefficiencies, as the design becomes memory-bounded since the memory interface cannot deliver data at the rate the single engine can process, resulting in underutilized hardware. Moreover, the control mechanisms, similar to those of a CPU, often cause additional inefficiencies.

Single engine architectures on FPGAs strike a balance between flexibility and efficiency, providing a scalable and reusable solution for a wide range of DNN workloads. Their

ability to execute DNN layers sequentially and adapt to various models makes them a versatile choice, though their memory-bounded nature highlights the trade-offs inherent in their design.

Streaming Architectures

Streaming architectures represent a distinct paradigm for FPGA-based acceleration, designed to achieve high throughput and efficiency by leveraging deeply pipelined hardware architectures. Unlike single engine architectures that reuse the same hardware for each layer, streaming architectures map the layers of a CNN model directly onto the FPGA fabric. This creates a dedicated, custom hardware block for each layer (or a small group of layers), as shown in Figure 2.12, forming a coarse-grained pipeline that mirrors the network's structure. The computations of each layer are overlapped via this deep pipelining. As one piece of data (e.g., the first few rows of a feature map) finishes processing in the hardware block for Layer N, it is immediately "streamed" to the hardware block for Layer $N + 1$. At the same time, the hardware for Layer N begins processing the next piece of incoming data. This allows multiple layers to be computationally active simultaneously on different parts of the data. This high degree of overlap is what ensures data flows continuously through the hardware, minimizing the need for repeated off-chip memory accesses [65]. Only the input to the first layer and the output of the final layer interact with the off-chip memory, while intermediate results are passed directly between consecutive layers within the architecture. This greatly reduces off-chip memory bandwidth requirements, addressing a major limitation of single engine architectures. However, this advantage comes at the cost of increased on-chip memory usage, as the pipeline requires the parameters for each layer to be present in the on-chip memory. Efficiently managing this on-chip memory is one of the primary challenges of implementing streaming architectures.

Streaming architectures are inherently model-specific, with each hardware block optimized for the unique computational requirements of its corresponding CNN layer. This customization allows streaming architectures to exploit layer-specific parallelism and achieve state-of-the-art performance for fixed models. However, the trade-off for this efficiency is reduced flexibility; streaming architectures are typically tailored to a specific application or model and are less suited to handling diverse workloads. Additionally, generating these architectures often involves lengthy compilation processes, as a new bitstream must be created for each target model since a dedicated hardware architecture is created for each specific model. Consequently, the hardware is not reusable across

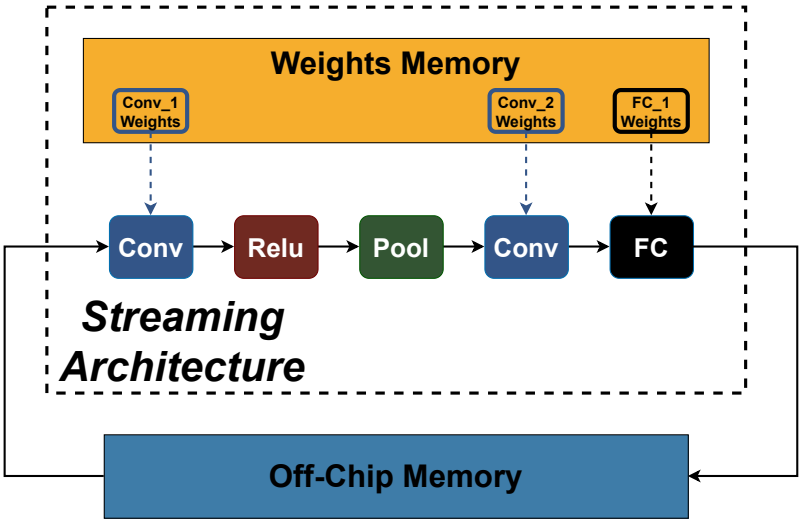


Fig. 2.12 Streaming architecture for FPGA-based DNN acceleration. The pipeline processes data across dedicated hardware blocks optimized for individual DNN layers (e.g., convolution, activation, pooling), leveraging layer-specific parallelism. Weights are managed in on-chip memory, while off-chip memory handles the data transfer to and from the first and last layers.

different models, and any change to the network architecture requires a full recompilation to generate a new bitstream.

Recent advancements have focused on addressing these challenges and enhancing the performance of streaming architectures. For example, PASS [66] leverages data sparsity by skipping zero multiplications, a technique controlled by either compile-time or run-time scheduling, while ATHEENA [67], introduces a toolflow for automating the mapping of early-exit networks to FPGA platforms.

Several automated toolflows have been developed to address the complexity of designing streaming architectures and optimize their performance. Tools like FINN [23], fpgaConvNet [24], DeepBurning [68], HLS4ML [26], FlexCNN [69], and HPIPE [25] enable the automated generation of streaming architectures. These frameworks have introduced various innovations, such as binarized neural network (BNN) optimizations in FINN, mixed-precision quantization in DeepBurning, weight sparsity exploitation in HPIPE skipping zero multiplications to improve performance, and multi-bitstream architectures in fpgaConvNet. Such toolflows emphasize optimizing computation engines and integrating efficient memory management techniques, including data buffering and streaming interfaces, to maximize throughput and minimize on-chip memory overheads.

While the bespoke nature of streaming architectures limits their flexibility, it also unlocks significant performance gains, making them a powerful solution for accelerating fixed DNN models. By implementing a pipeline of heterogeneous hardware blocks tailored to specific layers, these architectures offer unparalleled efficiency and throughput for deep learning tasks on FPGA platforms. Nonetheless, the challenges of increased on-chip memory demands and longer compilation times highlight the trade-offs involved in adopting this architectural approach.

fpgaConvNet Toolflow

The foundational concepts introduced by fpgaConvNet, such as Synchronous Dataflow (SDF) modelling, design space exploration (DSE), and multi-bitstream architectures, have served as the basis for the methodologies and strategies proposed and developed in this thesis. This subsection highlights fpgaConvNet's pivotal role as a precursor to this work, establishing how its principles have shaped and inspired the advancements presented in FPGA-based deep learning acceleration.

The fpgaConvNet toolflow [70, 71, 24] is a comprehensive framework designed to automate the process of mapping CNNs onto FPGAs. The primary inputs to the toolflow are a high-level description of a trained CNN model (typically in ONNX format), the hardware specifications of the target FPGA platform, and the optimisation target (e.g. throughput, or latency). The main output is a synthesizable hardware design, generated as High-Level Synthesis (HLS) C++ code, which can then be compiled into a bitstream for FPGA deployment. A high-level description of the toolflow, its different parts and the connections between them is show in Figure 2.13.

FpgaConvNet employs a streaming architecture that assigns dedicated hardware blocks to each layer of the CNN, constructing a coarse-grained pipeline that reflects the model's dataflow. A key innovation that distinguishes fpgaConvNet is the introduction of multi-bitstream designs, which allows it to partition large CNN models across multiple hardware configurations and overcome the challenges of mapping entire models onto a single FPGA configuration.

At the core of fpgaConvNet lies the Synchronous Dataflow (SDF) [72] model, a computational framework widely used for parallel systems. The SDF model facilitates accurate representation and optimisation of CNN workloads. Under this paradigm, CNN layers are represented as nodes in a directed acyclic graph, with arcs representing data streams. A-priori knowledge of data production and consumption rates, obtained via analytical performance modelling for each layer, enables static scheduling, hence

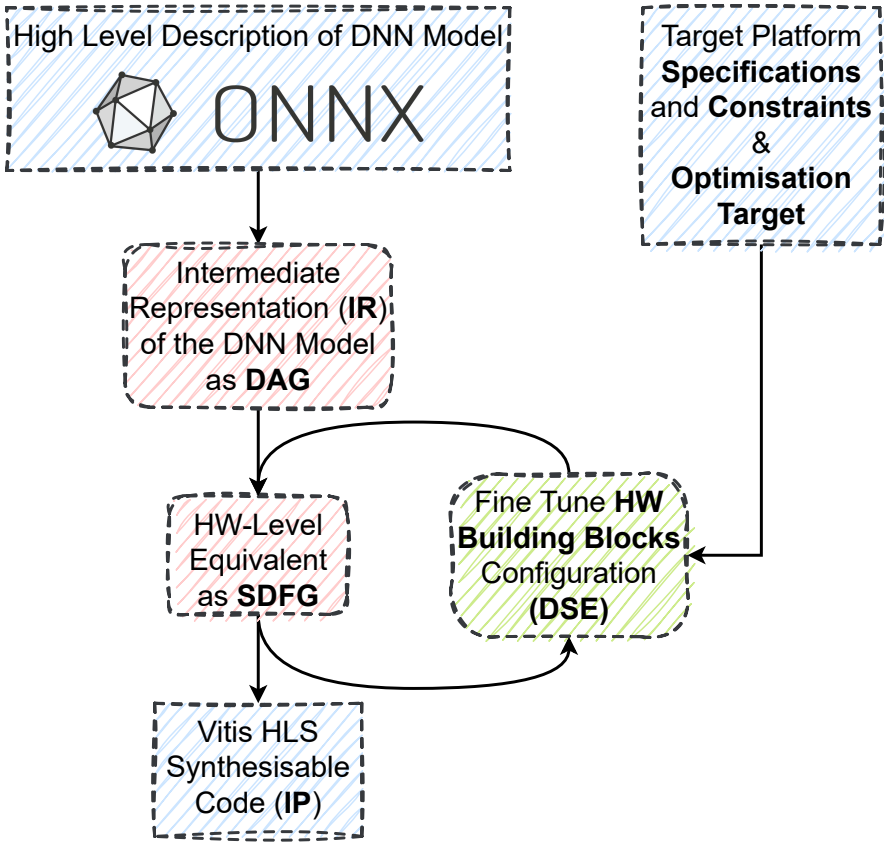


Fig. 2.13 The flow starts with a high-level description representation of the neural network model, converted to an intermediate DAG representation (IR). The IR is mapped to a hardware-level SDFG, with Design Space Exploration (DSE) optimizing hardware configurations based on platform constraints and specifications. The process generates synthesisable Vitis HLS code for FPGA deployment.

permitting performance optimisations at compile time. This systematic approach enables fpgaConvNet to explore the design space through algebraic transformations, tuning performance-resource trade-offs to produce efficient hardware mappings.

From an HLS perspective, the toolflow is built upon a library of highly configurable C++ templates, with each template corresponding to a specific CNN layer type (e.g., convolution, pooling, activation). The DSE process determines the optimal configuration for these building blocks by adjusting their compile-time parameters, such as parallelism factors (coarse-grain and fine-grain folding), to find the best performance-resource trade-off for the specific model and target device.

In the multi-bitstream design methodology, CNN models are partitioned into segments, with each segment executed on a specific hardware configuration. This division is especially advantageous for complex and deep models that exceed the resource utilisation

of a single FPGA device. Leveraging FPGA reconfigurability, fpgaConvNet supports larger models that would otherwise be impossible. Data flows sequentially through these configurations, with the FPGA being reconfigured between segments. Although this results in a time penalty from the reconfiguration overhead, the penalty can be amortized by employing larger batch sizes, making the approach suitable for throughput-oriented applications.

Another important characteristic of fpgaConvNet is its ability to balance performance and resource utilisation via iterative design space exploration (DSE). The DSE process systematically applies transformations on the Synchronous Dataflow Graph (SDFG), tweaking the parallelisation of each CNN layer while continuously optimising the model's overall performance. This iterative method ensures that total resource utilisation stays within the limits of the FPGA device. The exploration continues until the performance criteria are satisfied or the optimisation method converges, ensuring an efficient and resource-aware hardware implementation.

2.3 Human Action Recognition

Human Action Recognition (HAR) is a research area in computer vision, concentrating on the detection and categorisation of human actions from video data. It entails understanding the complex and dynamic motion patterns that define motions like walking, running, or gesturing by analysing spatiotemporal features from visual inputs [73–75]. HAR has been used in a variety of applications across various domains. In healthcare, it is utilised for patient monitoring, rehabilitation, and elderly care [76], where movement recognition can facilitate fall detection or recovery progress tracking. In smart surveillance systems, HAR improves security by detecting abnormal behaviours [77]. In sports analytics, it offers significant insights by autonomously categorising and analysing player activities [78, 79]. The advancement of HAR systems depends on advanced machine learning methodologies, especially deep learning models. Due to the increasing demand for processing 3D data, including video streams and volumetric data from sensors, 3D CNNs have emerged as the primary method for extracting spatial and temporal features, hence enhancing the capabilities of HAR. This section explores the architectural styles, methodologies, and evaluation metrics in HAR systems.

2.3.1 Human Action Understanding

The categorisation of human activities based on their complexity spans four key categories: gestures, actions, interactions, and group activities. Gestures involve fundamental movements of individual body parts, such as waving or pointing. Actions represent more complex behaviours, like walking or jumping. Interactions require the participation of at least two individuals, involving coordinated or responsive actions or interactions between an individual and an object, while group activities involve collaborative or simultaneous actions performed by larger groups of people. Human action understanding encompasses a range of tasks [75], including action recognition, temporal action localization, spatio-temporal action detection, and action prediction, each addressing different aspects of interpreting human behaviours in dynamic contexts. The ordering of these tasks also reflects an increasing level of complexity and challenge, driven by the need for more sophisticated solutions that integrate and combine multiple models and modules to capture, localise, and predict the human actions.

- The primary goal of **Action Recognition** is to accurately classify human actions to a predefined set of action classes within short, trimmed video clips, typically lasting only a few seconds. Trimmed videos are pre-processed video segments that focus exclusively on the target actions of interest, excluding irrelevant background activity. Various approaches and models have been proposed in the literature to tackle this challenge, including Video Masked AutoEncoders [80], X3D [6], Video Swin Transformer [81], and other state-of-the-art methods like UniFormerV2 [82], TSM [83], and SlowFast Networks [42] that are based either on vision transformers or 3D CNNs.

- The task of **Temporal Action Localisation (TAL)** entails localising the temporal boundaries and classifying actions within each instances in long, untrimmed videos, which can span several minutes. Untrimmed videos include not only the target actions but also unrelated background activity. To tackle this complex problem, researchers have introduced novel approaches such as ActionFormer [84], TALLFormer [85] utilising vision transformers, and R-C3D [86] employing 3D CNNs.

- The task of **Spatio-Temporal Action Detection** introduces the additional challenge of precisely localising the specific regions within images where actions occur in each frame. This involves identifying both the spatial location and the

temporal occurrence of actions in a video clip, as well as classifying the actions themselves. It requires accurate detection of the actors in the scene and their corresponding actions. Notable contributions in this field include works such as Tube Convolutional Neural Network (T-CNN) [87], Actor-Context-Actor Relation Network [88], MovingCenter Detector (MOC-detector) [89], and STEP [90].

- **Action prediction/anticipation** focuses on the task of predicting the expected outcomes of human actions using temporally incomplete video data. This involves anticipating the future actions or events based on the observed actions up to a certain point in time. Notable references in this area include ImagineRNN [91], Rolling-Unrolling LSTM [92], and Time-conditioned One-shot Anticipation using Attended Temporal Feature and Skip Connection (TOS AF SC) [93].

All modern approaches for addressing the four primary categories of human action understanding rely on features extracted by Deep Neural Network (DNN) models. These models serve as a foundation, providing rich spatiotemporal representations that are subsequently used for various downstream tasks. This thesis focusses on Human Action Recognition (HAR), however the backbone architectures and methodologies explored for feature extraction are versatile, making them applicable to other categories of human action understanding.

2.3.2 Architecture Styles for HAR

The remarkable success of Convolutional Neural Networks (CNNs) and vision transformers (ViTs) in computer vision tasks has motivated researchers to explore their application to video understanding. However, the utilisation of either CNNs or ViTs in video analysis poses several challenges that need to be addressed. These challenges include incorporating the temporal dimension, managing the increased complexity and memory requirements of algorithms, obtaining large-scale datasets with accurate annotations, accommodating the need for additional processing power, and coping with longer development cycles. In spite of these challenges, numerous studies have pushed the limits and attained remarkable and highly promising outcomes. To accommodate the extra temporal dimension, traditional CNN and ViT models have been extended with specialised modules or architectures capable of capturing and analysing this additional dimension. The main Deep Neural Network (DNN) architecture styles that have emerged for the task of HAR, can be classified into the following categories:

- **2D CNNs with Temporal Modelling:** One well-studied approach is the incorporation of temporal information in video analysis through the utilisation of 2D CNNs with temporal modelling techniques. Early works explored the use of LSTMs on top of 2D CNNs to aggregate and capture temporal information in videos [94, 95]. Another approach is the utilization of temporal pooling as an alternative solution to address the problem. These methods often introduce specialized or irregular building blocks within 2D CNN architectures to capture temporal dynamics [83, 96]. While these models are computationally and memory-wise less demanding compared to 3D CNNs, their performance tends to be inferior to them.

- **3D CNNs:** The utilisation of 3D CNNs provides a direct approach for capturing temporal information in video analysis. By applying 3D convolutions to stacked frames sampled from a video, these networks can effectively model the motion dynamics over a specific time period. Notable works, such as R(2+1)D [97] and SlowFast [42], have demonstrated impressive performance with such models. However, they often come with a significant computational and memory cost due to their large number of parameters and heavy workloads. To address these issues, researchers have started to focus on developing more lightweight 3D CNN architectures, such as X3D [6] and MoViNets [98], which offer comparable or even superior performance with reduced computational requirements. Despite these advancements, the computational and memory demands of even lightweight 3D CNNs remain high, posing challenges for real-time applications.

- **Vision Transformers:** Building upon the remarkable success of the Transformer [10] architecture in the computer vision domain [99], Bertasius et al. [100] have extended its application to the video domain proposing a convolution-free approach based solely on self-attention over space and time. This breakthrough has sparked significant interest, leading to recent works that focus on vision transformers and achieve promising and state-of-the-art results, as demonstrated in various works such as Liu et al. [81], and Fan et al. [101]. However, it is important to note that the impressive performance of vision transformers comes at a significant computational cost. Most of these models require extensive computations, which may limit their practical deployment in real-world scenarios. For instance, the TubeViT-L [102], a state-of-the-art vision transformer model, has a computational workload of approximately 794 million MACs. In comparison, the X3D-XXL [6], the largest model in the X3D family of 3D CNNs, requires only 194 million MACs—making the

workload of TubeViT-L nearly four times greater. Figures 2.14 and 2.15 provide further insights into the differences between vision transformers and 3D CNNs.

- **Multi-Stream Networks:** By employing multiple models, whether they are 2D, 3D or ViT, the goal is to capture both spatial and temporal information simultaneously. These architectures often consist of two separate streams: one stream focuses on spatial features using still images, while the other stream incorporates optical flow, motion vectors, or other semantic information to capture temporal features. The outputs from all streams are then fused to generate the final classification results. While these models have achieved impressive performance, they pose challenges in terms of computational complexity, and resource requirements. Incorporating additional semantics alongside RGB information in multi-stream architectures can provide performance improvements for 2D, 3D, and ViT models. However, this enhancement comes at the expense of increased computational complexity and memory requirements.

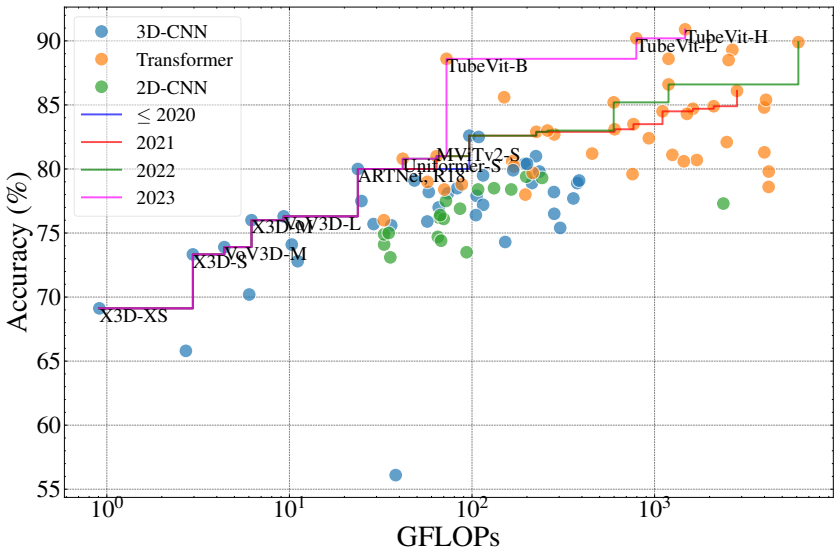


Fig. 2.14 The Pareto front of Top-1 Accuracy (%) versus GFLOPs, over the years (represented by four coloured lines) on Kinetics-400 benchmark.

Figures 2.14 and 2.15 showcase the increased and ongoing interest, as well as the rapid evolution and advancements of neural network model architectures for the vision-based human action recognition task. The Pareto front of accuracy over GFLOPs evolves and pushes forward every year, as evidenced by the four coloured lines (blue, red, green, and magenta) in two of the most widely used evaluation benchmarks, Kinetics-400 [103] and

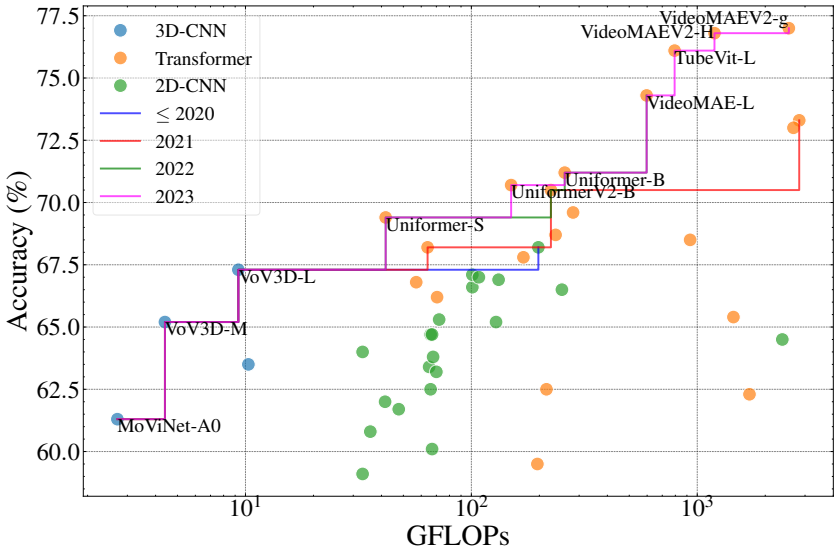


Fig. 2.15 The Pareto front of Top-1 Accuracy (%) versus GFLOPs, over the years (represented by four coloured lines) on Something-Something-V2 benchmark.

Something-Something-V2 [104]. The bottom-left part of the Pareto is dominated by 3D CNN models, whereas the top-right one is dominated by vision transformer [99] networks. This provides a solid understanding of the characteristics of the two architecture styles. On one hand, Vision transformers accomplish higher levels of accuracy but at the expense of a significantly increased computation cost in GFLOPs. 3D CNNs, on the other hand, strike a remarkable balance between performance and computational cost, providing solutions with orders of magnitude fewer GFLOPs, encompassing both memory and execution time considerations. Among other HAR benchmarks, the Kinetics-400 and Something-Something-V2 datasets presented here are further discussed and analysed in the following Section 2.3.3.

2.3.3 Evaluation Process and Public Benchmarks

The evaluation of Human Action Recognition (HAR) models begins with the conversion of unprocessed video data into a structured input format compatible with the neural network's input. Given a video comprising a continuous sequence of M frames, K frames are sampled using a sampling rate R to represent the spatiotemporal content of the video. This sampling method aims to preserve essential temporal information as accurately as possible while reducing computational complexity. Figure 2.16 illustrates the aggregation of the sampled frames to form a compact representation, which serves as the input (clip) to the 3D CNN model.

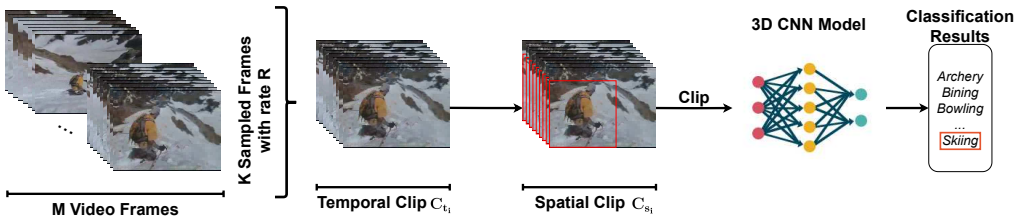


Fig. 2.16 The process of converting raw video data into a structured input format for 3D CNN models. The Figure illustrates the process of both temporal and spatial clipping that form the final input clip to the model.

The sampling strategy and the total number of sampled frames are critical components of the pre-processing of input video data. The sample rate affects the capacity of the model to interpret longer videos, as sparse sampling may neglect important temporal dependencies. On the other hand, a denser sampling increases the model's computational complexity owing to the increased number of frames at the model's input. Consequently, these parameters are carefully examined in the design and optimisation of HAR models from a machine learning perspective, as they represent important trade-offs between accuracy and efficiency. This thesis is not focused on investigating or improving sampling strategies or frame selection; rather, it considers those as fixed derived from the original model architectures used in the experiments.

The deployment of HAR models typically involves generating multiple new video clips from a single input video. A "clip" in this context refers to a shorter segment of the original video, defined by temporal cropping, spatial cropping, or a combination of both. First, a set of temporal clips (C_t) is created by shifting a fixed time window throughout the duration of the original video. Additionally, a set of spatial clips (C_s) is generated by cropping different areas from each frame of the original video. This results in a total of $|C_t| \times |C_s|$ clips, each of which is passed individually through the HAR model. The final prediction for the original video is obtained by averaging the predictions of all $|C_t| \times |C_s|$ clips. It is common that the evaluations use a fixed number of 10 temporal and 3 spatial clips per input, so the total computational cost scales proportionally, making the inference-time during the evaluation of the model cost 30 times the MACs required for a single clip.

3D CNN models accept as input the clip(s) structured as a tensor with dimensions of $[B, F, C, H, W]$, where B denotes the batch size (e.g. the total number of clips $|C_t| \times |C_s|$), F is the number of frames per clip, C represents the number of channels (e.g., 3 for RGB), and H and W correspond to the height and width of each frame in each clip.

This structured format allows the model to extract both spatial and temporal features, enabling accurate recognition of dynamic actions over time.

HAR Benchmarks

Human Action Recognition (HAR) has been extensively evaluated using a variety of widely adopted benchmarks. Among the most prominent ones is the Kinetics [103] dataset, available in three versions, Kinetics-400, Kinetics-600, and Kinetics-700, that provides a large-scale collection of video clips annotated with diverse action labels, making it a standard benchmark for general action recognition tasks. The Something-Something [104] dataset, with its two versions (V1 and V2), focuses on fine-grained action recognition involving object interactions. UCF101 [1] and HMDB51 [105] are traditional benchmarks consisting of relatively smaller datasets that have played a pivotal role in early HAR research, offering a mix of actions performed in real-world scenarios. Moments-in-Time [106] provides a large-scale dataset for recognizing short actions and events, generalizing across a wide variety of categories involving people, animals, objects and even natural phenomena. Charades [107] introduces a multi-label dataset that focuses on everyday activities captured in home settings. Lastly, the AVA (Atomic Visual Actions) [108] dataset, available in two versions (V2.1 and V2.2), is tailored for spatio-temporal action detection, requiring models to identify and localize actions at the frame level.

Evaluation Metrics for HAR Benchmarks

Evaluation metrics play a crucial role in assessing model's performance on these benchmarks. For Kinetics, Something-Something, UCF-101, HMDB51, and Moments-in-Time datasets, **Top-N** accuracy is the primary metric, typically focusing on Top-1 and Top-5 accuracies. *Top-1* measures how often the model's top prediction matches the ground truth, while *Top-5* evaluates if the correct label is among the top five predictions of the model, addressing cases with multiple plausible actions. For Charades and AVA datasets, **Mean Average Precision (mAP)** is used, as it evaluates precision-recall performance for each class while considering classification accuracy and temporal or spatial localization. mAP is particularly suited for multi-label tasks and spatio-temporal detection, offering a more detailed assessment of model performance.

2.4 Overview of 3D CNN HAR Accelerators

The acceleration of 3D CNNs on FPGA platforms has drawn research interest in recent years, due to increasing demands for high-performance solutions in tasks like human action recognition (HAR). However, this topic remains relatively under-explored compared to 2D CNN accelerators, with a limited number of works targeting 3D CNNs explicitly. This section reviews the state-of-the-art 3D CNN FPGA accelerators, discussing their limitations, and highlighting how this thesis addresses these gaps.

2.4.1 Early Efforts and Advanced Optimisation Strategies

Early Efforts in FPGA-Based 3D CNN Acceleration:
Most early works in this field have focused on relatively outdated models such as the C3D [3], whose simplistic architecture, which can be compared to the architecture complexity of AlexNet [11] in the 2D domain, limits its performance compared to modern state-of-the-art models such as X3D [6] and SlowFast [42]. For instance, Fan et al. [109–111] introduced a series of works on 3D CNN acceleration for HAR on FPGAs. In their initial work [109], they proposed F-C3D hardware architecture for the acceleration of C3D [3], which is capable of supporting multiple 3D convolutional layers and design strategies for overcoming the challenges associated with 3D CNNs, such as increased computation and memory requirements, while also allowing their design to be ported to other FPGA devices. In their subsequent work [110], they extended this with an analytical model and a tool for optimising the hardware architecture based on the device specification and accuracy requirements, as well as the use of block floating point (BFP) arithmetic precision to minimise accuracy loss and the need for retraining the model. In their most recent publication [111], they proposed E3DNet, an efficient 3D CNN based on their proposed 3D-1 bottleneck building block. Their hardware implementation of E3DNet, called F-E3D, is capable of real-time performance at a latency of 35.3 ms per clip, while achieving 85.1% Top-1 accuracy in the UCF101 [1] benchmark. While effective this approach falls short in terms of accuracy in UCF101, over 13*pp*, compared to the current state-of-the-art model by H. Duan et al. [112].

Liu et al. [113] similarly targeted the C3D model, proposing a unified hardware architecture for 2D and 3D CNN acceleration based on their observation that the computing patterns of 2D and 3D CNNs are similar. Their aim was to convert the CNN convolutions to matrix multiplication operations, paying close attention to memory

optimisations in order to overcome the difficulties of feature map replications. Additionally, they employed an analytical model to configure the accelerators for optimal resource use. Shen et al. [2] followed a similar approach, developing a unified template-based architecture based on the Winograd [114] algorithm capable of handling both 2D and 3D CNNs. Additionally, they developed an analytical technique for efficiently exploring the design space for mapping 2D and 3D CNNs on FPGA accelerators. Both approaches, however, were limited to the relatively simple C3D model and did not evaluate more modern and computationally intensive architectures.

Advances in Optimisation and Design Space Exploration:
T. Teng et al. [115] proposed a design space exploration strategy for optimising memory access in 3D CNN models accelerated on FPGAs. The authors proposed a non-overlapping data tiling method for off-chip memory access and explored on-chip data reuse using different loop ordering strategies. They further proposed a hardware architecture that can support these strategies. Their work achieved state-of-the-art performance on the C3D model but did not evaluate more complex architectures. Sun et al. [116] used a blockwise pruning approach to apply weight pruning to two distinct 3D CNN architectures, namely C3D and R(2+1) [97]. Their hardware design, which is based on the Alternating Direction Method of Multipliers (ADMM), together with the suggested pruning approach, enables the acceleration of 3D CNNs with low accuracy loss when compared to their unpruned form. Khan et al. [117] extended the scope of 3D CNN acceleration to more modern architectures, targeting the I3D [41] model, a 70-layer deep neural network. They adjusted the feature-map word lengths and weights in a pre-trained model, which reduced its complexity without affecting its accuracy. They proposed a data tiling technique that utilises all four dimensions of video data and improves memory bandwidth while reducing DRAM accesses. Based on these optimisations, the proposed FPGA accelerator achieves 684 GOPs/s for 32-bit floating point and 1.29 TOPs/s for 8-bit integer implementations with a 2% accuracy drop.

To further elaborate on the discussion of state-of-the-art works, Figure 2.17 illustrates the Pareto front of Top-1 accuracy (%) versus throughput (clips/s) for FPGA-based 3D CNN accelerators on the UCF101 HAR benchmark. This visualisation highlights the trade-offs achieved by various approaches, where marker colours denote the targeted 3D CNN models, and marker shapes represent the FPGA platforms.

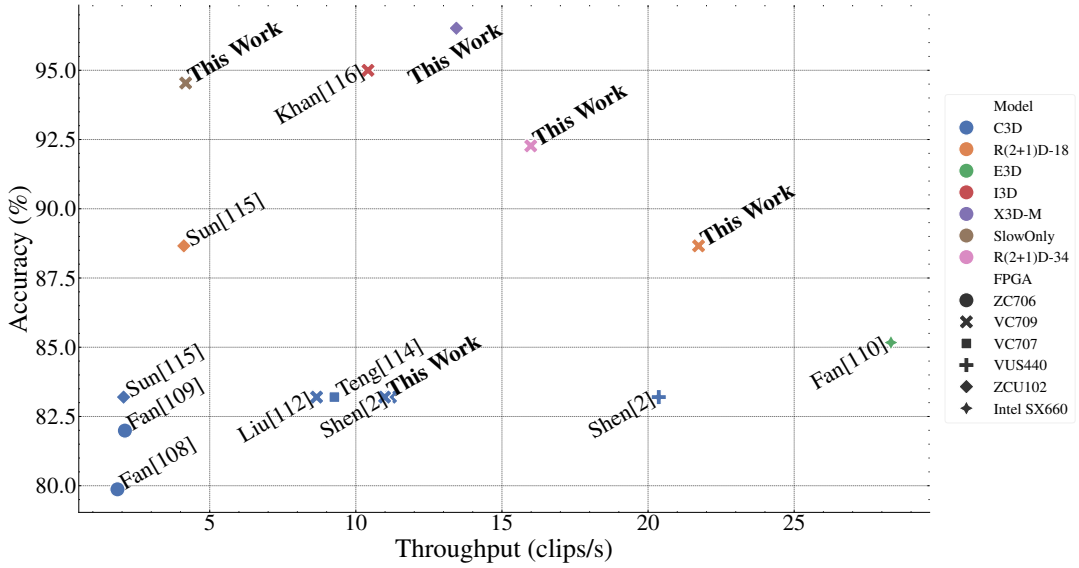


Fig. 2.17 Pareto front of Top-1 Accuracy (%) versus Throughput (clips/s) for FPGA-based 3D CNN accelerators on the UCF101 [1] HAR benchmark. Marker colours indicate the 3D CNN models targeted in each work, while their shapes represent the FPGA platforms used. The C3D model used by Shen et al. [2] is a different and smaller version from the original one [3] used by the rest of the works.

2.4.2 Limitations of Existing Approaches

The majority of existing works have focused heavily on the C3D [3] model, introduced in 2013, and whose architecture comprises of eight convolutional layers and lacks the complexity and depth of modern state-of-the-art HAR models. C3D's relatively low accuracy (85.2% on UCF101 compared to the current state-of-the-art's 98.6%) highlights the limitations of these designs in meeting the demands of modern applications. Moreover, the focus on rather simplistic model architectures has left a gap in addressing the challenges posed by more advanced 3D CNNs, such as X3D [6] and SlowOnly [42], which feature significantly deeper and more complex architectures. These models can span up to 200+ layers deep, having complex structures like branches and even nested branches, as seen in X3D. Additionally, their skip connections can reach up to 15 layers, adding further challenges to their design optimisation and memory management.

Most of the existing approaches are model-specific, with designs tailored to individual 3D CNN architectures such as C3D, R(2+1)D, or I3D, raising concerns about their scalability and adaptability to other models. It remains unclear how these solutions could generalize to support a broader range of architectures. Additionally, there is a notable lack of toolflows capable of automating the mapping and optimisation of diverse 3D

CNN models across various FPGA platforms, posing a significant challenge to the wider adoption and practical deployment of FPGA-based 3D CNN accelerators in real-world scenarios.

2.4.3 Contributions of This Thesis

This thesis addresses the limitations of existing approaches by presenting significant advancements in the design and deployment of FPGA-based 3D CNN accelerators. While prior works have mainly focused on specific models like C3D, this research introduces a toolflow capable of supporting a broad spectrum of 3D CNN architectures. Notably, it targets modern state-of-the-art models such as SlowFast (2020) and X3D (2020), delivering improvements of 20.35*pp* and 21.4*pp* in Top-1 accuracy on the Kinetics-400 benchmark, respectively, compared to C3D. The proposed toolflow enables automated mapping and optimisation across different FPGA platforms, facilitating scalability and flexibility, while being able to produce either throughput (Chapters 4 and 5) or latency (Chapter 6) oriented designs. Furthermore, this thesis presents novel design techniques that achieve a balance between throughput/latency, and FPGA resource utilisation. These advancements are driven by a finely-tuned design space exploration, that thoroughly explored the trade-offs between performance and resources, providing optimal solutions. By focusing on a more generalisable and automated solution, this research expands the applicability of FPGA-based accelerators to diverse and more complex CNN architectures, establishing a robust foundation for high-performance and CNN acceleration in real-world applications.

Chapter 3

Embarking on the Acceleration Journey: A Primer on FPGA-Based 3D CNN Model Optimisation

3.1 Introduction

In this chapter, we investigate the optimisation of mapping and deploying 3D Convolutional Neural Networks (3D CNNs) on Field-Programmable Gate Arrays (FPGAs), with an emphasis on addressing their inherent computational and memory challenges. Building upon the foundational understanding of 3D CNNs established in previous chapters, we delve into the specifics of mapping X3D, a state-of-the-art model in Human Action Recognition, onto FPGA devices. Achieving a remarkable 95.5% accuracy on the UCF101 [1] benchmark, the X3D model is an excellent example of the potential and complexity of 3D CNNs. Our proposed solution generates a streaming-based hardware architecture of X3D that takes into account the target FPGA's available resources, specifically optimized to maximize throughput. While this chapter focuses on targeting the functional blocks of the X3D model, the proposed approach can be considered as a more generic tool. In the subsequent chapters, this work will expand its scope to address a broader range of 3D CNN models, demonstrating its versatility and adaptability. The work of this chapter pioneers the deployment of sophisticated 3D CNN architectures for Human Action Recognition on FPGA platforms, thereby opening new avenues for real-world applications in resource-constrained environments.

In recent years, two-dimensional Convolutional Neural Networks (2D CNNs) have demonstrated exceptional performance in image-related tasks. However, the growing focus on video-related applications, such as video surveillance, autonomous vehicles, and patient monitoring, necessitates the development of algorithms that take into consideration the temporal domain. Three-dimensional CNNs (3D CNNs) are regarded as a viable approach for dealing with video and volumetric data, as they can cope with an extra dimension, such as time or depth, and enhance their learning capacity by extracting features linked to this additional dimension.

In the specific task of human action recognition (HAR), 3D CNNs have demonstrated exceptional performance. By interpreting human motion across video frames, 3D CNNs can detect a variety of human actions without the need for dedicated time-domain techniques like Long Short-Term Memory networks (LSTMs) [118]. As illustrated in Figure 3.1, 3D CNNs dominate the Pareto front in the widely-used HAR benchmark, Kinetics-400 [103], although the recent emergence of vision transformers is beginning to challenge this dominance. However, these transformers architectures require orders of magnitude more GFLOPs to operate, a trade-off that can only be tolerated in scenarios with access to substantial computational resources.

Despite their capabilities, the additional dimension in 3D CNNs often results in even greater computational and memory demands compared to 2D CNNs. To mitigate the high processing requirements of both 2D and 3D CNNs, various hardware devices such as GPUs, FPGAs, and ASICs have been employed. FPGAs, in particular, stand out as an attractive acceleration platform due to their flexibility, energy efficiency, and adaptability to rapidly evolving 2D and 3D CNN model designs.

Previous FPGA-based acceleration efforts for 3D CNN models have primarily focused on models like C3D and R(2+1)D, which either have not been evaluated or fall short of the Pareto front in recent benchmarks such as Kinetics-400, as show in Figure 3.2. In contrast, this chapter presents an FPGA architecture for X3D, a 3D CNN model that sits on the Pareto front of accuracy versus the number of parameters. The proposed approach building upon and expanding fpgaConvNet, employs an automated optimisation process, applying transformations to the Synchronous Data Flow (SDF) graph to explore the broad design space of X3D within a streaming architecture, tailored to the target FPGA platform's resources. The key contributions of this chapter include:

- Extending the SDFG model, as previously described in Section 2.2.3, to capture the performance requirements of mapping 3D CNNs onto streaming architectures by defining and incorporating X3D layers as hardware building blocks, with a

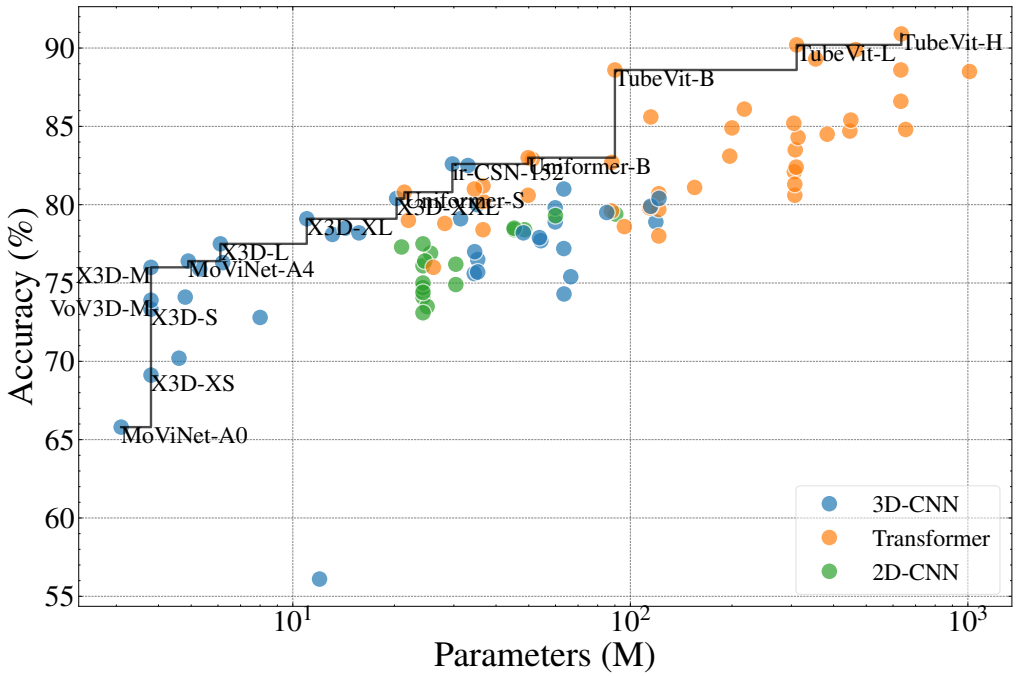


Fig. 3.1 Pareto front (performance over number of parameters) on the Kinetics-400 benchmark. The lower left side of the graph demonstrates that 3D CNNs achieve Pareto front solutions having orders of magnitude less parameters than Vision Transformer architectures.

particular focus on handling irregular blocks with branching, which are prevalent in modern 2D and 3D CNN models.

- Automating and optimising the mapping of the X3D model, a state-of-the-art model for Human Action Recognition, to various FPGA devices, with a throughput-oriented approach that accounts for available FPGA resources and memory bandwidth characteristics.

- Implementing model-specific optimisations to improve the dataflow characteristics of the X3D network, such as the use of prior batch statistics for global average pooling (GAP) layers, resulting in higher performance designs with minimum impact on the model's accuracy.

The additional dimension in the feature maps of 3D CNNs significantly increases the workload and computational complexity at each layer. As a result, current streaming-based FPGA mapping approaches are insufficient to address these challenges. Existing architectures, such as fpgaConvNet [24], HLS4ML [26], and FINN [23], have not tackled the unique demands of 3D CNNs, and adapting them is far from straightforward. A new

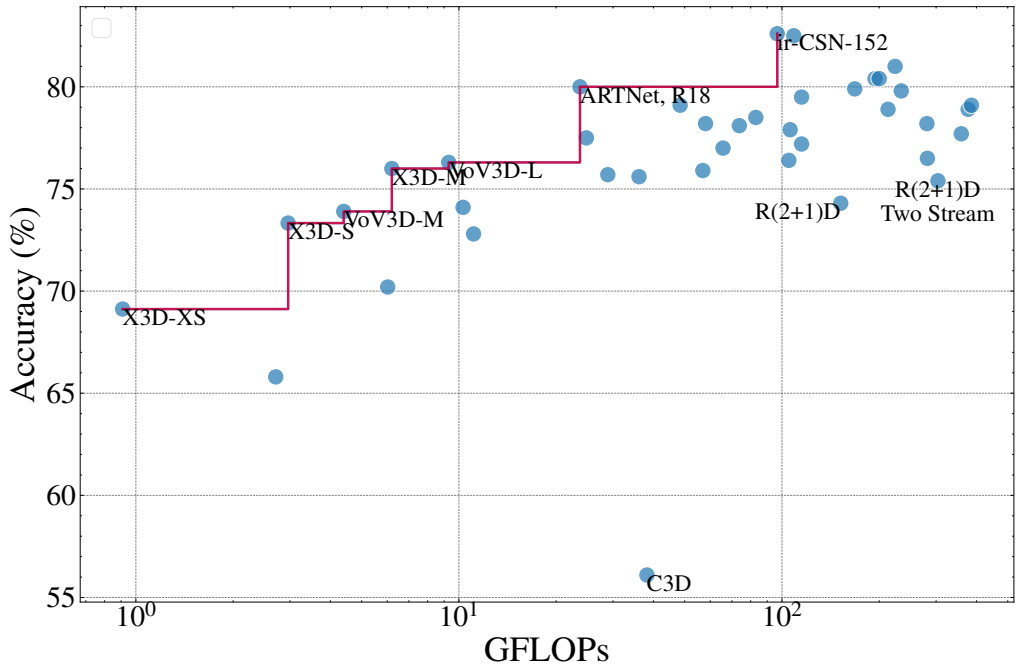


Fig. 3.2 Pareto front (performance over number of operations) on the Kinetics-400 benchmark for 3D CNNs.

hardware design is required to efficiently process 3D tensors and execute 3D dot products in a streaming fashion. This design must account for the increased on-chip memory and buffering needs, as well as the higher off-chip memory bandwidth requirements, which differ greatly from those of 2D CNNs, while ensuring scalability across the system. Section 2.2 provides a more in depth discussion of the differences between 2D and 3D CNN architectures. By addressing these challenges, this chapter aims to push the current performance-accuracy Pareto front and enable the deployment of sophisticated 3D CNN architectures for HAR on FPGA platforms, paving the way for their use in real-world, resource-constrained environments.

3.2 Background

To fully comprehend the innovations and optimisations introduced in this chapter, it is essential to dive deeper into the X3D model family. In this section, we will look at the fundamental concepts, architectural innovations, and performance benchmarks that define the X3D model family, laying the groundwork for understanding subsequent FPGA-based optimisation strategies. The X3D model family of efficient 3D CNN models was originally proposed by C. Feichtenhofer [6] in 2020. The authors have created various

model configurations by stacking different number of layers to create bigger blocks, following the ResNet [9] structure paradigm. The core block of the X3D models family is depicted in Figure 3.3. The model designs in the family range from X-Small (X3D-XS) to XX-Large (X3D-XXL), depending on the computational complexity, number of core layers, input volume size, and number of parameters. The computational complexity of the X3D-XS is 0.6 GFLOPs, whereas the computational complexity of the X3D-XXL is 143.5 GFLOPs. A more detailed view of the X3D model variants can be seen in Table 3.1 below. The table display the top-1 classification accuracy (percentage) alongside the computational complexity, measured in gigaflops (GFLOPs), which represents the number of floating-point operations (multiply-adds $\times$ 10^9) for a single clip input. The definition of a "clip" and the evaluation procedure followed for assessing HAR models are discussed in Section 2.3.3 of Chapter 2. The number of model parameters ranges from 3.76 M to 20.3 M. The X3D models have excelled in many HAR benchmarks, including the Kinetics-400 (80.0%), Kinetics-600 (81.9%), and Charades [107] (47.1%).

Table 3.1 Variations of the X3D model evaluated on the K400-val dataset using 10-centre clip testing. Original table information from [6].

Model	top-1 (%) @ K400-val	FLOPs (G)	Params (M)
X3D-XS	68.6	0.60	3.76
X3D-S	72.9	1.96	3.76
X3D-M	74.6	4.73	3.76
X3D-L	76.8	18.37	6.08
X3D-XL	78.4	35.84	11.0
X3D-XXL	80.0	143.5	20.3

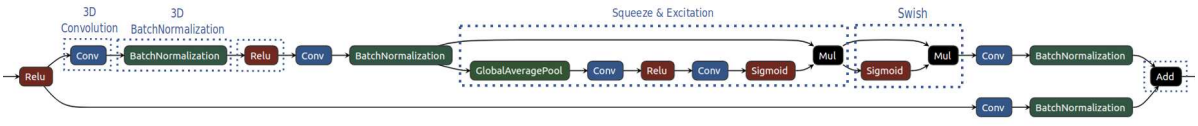


Fig. 3.3 X3D model main building blocks consisting of a series of 3D convolutions, 3D batch normalisation layers, non-linear activations, and specialised blocks like Squeeze and Excitation and Swish activations.

We have identified and analysed all the primitive modules that are necessary to build up the X3D blocks. As can be seen in Figure 3.3 these modules are the 3D Convolution (Depthwise and Pointwise), Activation operations (Sigmoid, Swish and ReLu), Squeeze and Excitation (which we defined as a primitive module, although it is a combination

of other modules), 3D Batch Normalisation, and any kind of element-wise operation like addition, multiplication, etc. More detailed information regarding these primitive modules have been provided in Sections 2.1.1 and 2.1.2, apart from the Squeeze and Excitation and Fully Connected, which we describe below:

- **Squeeze and Excitation.** Hu *et al.* [48] first introduced the idea of Squeeze and Excitation (SE) trying to better capture the relationship between channels and allow information to be passed from the global receptive field to all the layers. The input of this module is passed through the squeeze part, i.e. a Global Average Pooling module. This is followed by the excitation part, which comprises two Fully Connected (FC) modules accompanied by their activation operations where the former reduces the dimensionality while the later increases it back to the dimension of the squeeze operation. The final output of the SE is obtained by multiplying the result of the excitation part with the original input of the module with broadcasting, and so the dimensions of the output match those of the input. The number of parameters this modules holds depends only on the parameters of the two FC modules. In X3D models the FC modules are implemented as pointwise 3D convolutions with the weight matrix of the FC translated as the input and output channels of the 3D convolution.

- **Fully Connected.** In order to obtain the final classification, fully linked or dense layers are commonly utilised as the last layers of CNN models. Although their computational complexity is much lower than those of the convolutional layers, they hold a lot of parameters thus they require higher memory for the weights. Given an input (S^{in}) of shape $1 \times I$, and a weights matrix (W_{eights}) of shape $I \times O$, the output (S^{out}) shape of a FC operation would be $1 \times O$. Mathematically each value of the output can be calculated as follows:

$$\mathrm{S^{out}[o]} = \sum_{i=1}^{I} \mathrm{S^{in}[i]} * \mathrm{W_{eights}[o][i]} \tag{3.1}$$

  The computational complexity of the Fully Connected operation is equal to $2 \times I \times O$, while the number of learnable parameters is equal to $I \times O$.

The main focus of this part of the work is the X3D-M, a model with a computational complexity of 6.2 GFLOPs, 3.76 million parameters, that receives at its input a sequence of 16 RGB frames with spatial dimensions of 256×256. It achieves an accuracy of 76.0%

on Kinetics-400, 78.8% on Kinetics-600, and 96.5% on UCF101. X3D-M consists of 244 distinct layers, including depthwise and pointwise 3D convolutional layers, various activation functions such as Sigmoid, Swish, and ReLU, 3D global average pooling layers, and fully connected layers. We selected X3D-M from the family of X3D models because, while it shares the same type of layers and overall architecture as the other models in the X3D family, it offers a balanced trade-off between performance and computational workload. With a moderate number of layers, GFLOPs, and parameters, X3D-M strikes the right balance between accuracy and efficiency, making it better suited to the computational scale of our target FPGA platform. This middle-scale model aligns well with the hardware constraints while maintaining competitive performance in action recognition benchmarks.

3.3 FPGA-based Modelling of 3D CNNs

This following sections describe the proposed strategy for representing the 3D CNN-based X3D model architecture at intermediate and hardware-level representations, following the main principles and general flow structure of fpgaConvNet as detailed in section 2.2.3. The solution begins by taking as input the high-level description file of the X3D model in PyTorch or ONNX format, and capturing the parameters of each layer in a Directed Acyclic Graph (DAG) along with the connections between the layers. Each of the supported network layers is mapped to a hardware building block equivalent that implements its functionality at the hardware level. The Synchronous DataFlow Graph (SDFG) is then built for the network by mapping the DAG nodes to their hardware equivalents and adding them as nodes in the SDFG, as well as adding the connections between the DAG nodes as arcs. Finally, the expected performance of a specific configuration of the network's SDFG nodes is estimated using the Synchronous DataFlow (SDF) computation model. Venieris et al. [119] provide an excellent walk-through and a more in-depth, practical example of this methodology in their section ''Synchronous Dataflow Modelling for ConvNets", demonstrating how SDF parameters relate to performance and data rate balancing in a streaming architecture. A high-level description diagram of this flow is presented in Figure 2.13. The following sections provide detailed descriptions for each stage of the proposed solution's workflow.

3.3.1 DAG Representation

The translation of the high-level X3D model into an Intermediate Representation (IR) described as a Directed Acyclic Graph (DAG) is detailed here. This involves representing each layer and its connections as nodes and edges in the DAG, facilitating the subsequent mapping onto hardware.

A neural network model, as described by high-level frameworks such as PyTorch and ONNX, consists of three main parts. The first specifies the layers and their connections, which defines the model's structure and flow. The second details each layer's specific properties and configuration. The final holds the actual values of the learnable parameters, known as weights, that are associated with the layers (if applicable). A dedicated model parser is provided, that parses the above descriptions to build a DAG containing all of the relevant information about the target neural network. The DAG structure is faithful to the original, retaining just the essential information from the layers' specific attributes and configuration. Additionally, the parser stores the model parameters/weights for future use during inference.

To facilitate the hardware-level mapping of the model as described in the next section, it is essential to store some key parameters of the model's layers, such as the input and output feature-map sizes and certain layer-specific parameters. Feature-maps, which define the spatial, temporal, and channel dimensions of the data being processed by each layer, are represented as follows:

$$\boldsymbol{S_n^{in}} = \{H_n^{in}, W_n^{in}, D_n^{in}, C_n^{in}\}$$
$$\boldsymbol{S_n^{out}} = \{H_n^{out}, W_n^{out}, D_n^{out}, C_n^{out}\}$$

where $H_n^{in/out}$, $W_n^{in/out}$, $D_n^{in/out}$, and $C_n^{in/out}$ represent the height, width, depth, and number of channels for the input and output, respectively. The size of a feature-map, denoted as $|\boldsymbol{S}|$, defines the total number of elements it contains. These parameters, $\boldsymbol{S}_n^{in}$ and $\boldsymbol{S}_n^{out}$, form part of the parameter space definition for all the layers of the model, along with additional layer-specific configurations, as outlined below and they are essential for accurately mapping the network's operations to their hardware-level equivalent representations.

- **3D Convolutional Layers** There are two different types of convolutional layers that the X3D model incorporates: (a) 3D depthwise convolutions, (b) 3D pointwise convolutions. Even though the are different types of convolutions, the discrimination

between them can be achieved by looking at the layer's configuration that is stored in the DAG nodes, $< \mathbf{S_n^{in}}, \mathbf{S_n^{out}}, \mathbf{K_n}, \mathbf{J_n}, \mathbf{P_n}, \mathbf{Gr_n} >$:

- $\boldsymbol{K}_n$ is a 3-value vector $[K_n^h, K_n^w, K_n^d]$ specifying the height, width, and depth of the 3D convolution window (also known as kernel).
- $\boldsymbol{J}_n$ is a 3-value vector $[J_n^h, J_n^w, J_n^d]$ specifying the strides of the convolution along each dimension.
- $\boldsymbol{P}_n$ is a 6-value vector $[P_n^{h_s}, P_n^{h_e}, P_n^{w_s}, P_n^{w_e}, P_n^{d_s}, P_n^{d_e}]$ denoting the amount of padding applied at the start and end of each dimension of the input.
- $\boldsymbol{Gr}_n$ is a value that specifies the number of groups to which the input is split along the channel axis.

- **3D Activation Layers** The activation operations used in X3D are the following: (a) Sigmoid activation, (b) Swish activation which is expressed as $y = x * sigmoid(x)$. (c) ReLu activation,. The layer's configuration is as follows, $< \mathbf{S_n^{in}}, \mathbf{S_n^{out}}, \mathbf{T_n} >$:
  - $\boldsymbol{T}_n$ denotes the type of activation operation.
- **3D Element-wise Layers** In X3D, element-wise operations refer to the layers that combine data from several branches via addition or multiplication. In comparison to the other layer types, these layers combine several inputs into a single output. The shapes of the layer's inputs may or may not be identical, resulting in different functionality of the layer (normal vs broadcasting). The layer's configuration is as follows, $< \mathbf{S_n^{in1}}, ..., \mathbf{S_n^{inN}}, \mathbf{S_n^{out}}, \mathbf{T_n}, \mathbf{Q_n} >$:
  - $\boldsymbol{T}_n$ denotes the type of element-wise operation, e.g. addition, multiplication, division, and so on.
  - $\boldsymbol{Q}_n$ denotes the mode of element-wise operation, normal or broadcasting.
- **3D Global Average Pooling Layers** While the standard pooling operation samples patches of the input feature map to decrease its size, this extreme case samples the whole feature map into a single value, creating an output vector with the same shape as the channels, with the configuration being, $< \mathbf{S_n^{in}}, \mathbf{S_n^{out}} >$.
- **3D Batch Normalisation Layers** These are typically used in conjunction with 3D convolution. 3D Batch Normalisation has a modest number of parameters, equal to the number of channels in the layer's input $\times 4$ (one for each of the parameters

$\gamma, \beta, E[x], Var[x]$). The configuration of a batch normalisation layer is the following, $< \mathbf{S_n^{in}}, \mathbf{S_n^{out}} >$.

- **Fully Connected Layers** The Fully Connected layers are transformed into Point-wise Convolutional Layers, with the FC kernel size converted into the output channels of the Convolutional layers.

3.3.2 Converting to Hardware Building Blocks

This section describes the process of converting the IR of the layers within the Directed Acyclic Graph (DAG) representation to hardware building blocks, creating hardware equivalent nodes for each layer of the X3D model as part of the SDFG. To take advantage of the SDF model's capabilities, we map DAG nodes to their corresponding hardware building blocks, which implement the functionality of the respective network layers in the underlying hardware. This section also delves into data flow optimisation and resource allocation, demonstrating how these parametrised hardware building blocks could impact the model's efficient execution and utilisation of hardware resources.

The hardware building blocks are the major components utilised to construct the SDFG, denoted as G with N computation nodes, and $G = \{n_1, ..., n_N\}$, which will be used subsequently to estimate the network's performance. The configuration of these blocks, in conjunction with the network's topology, is utilised to automatically generate and construct the design's synthesisable Vitis HLS code, as highlighted in Figure 2.13. The representation of the supported hardware building blocks comprises the following:

i) **DAG parameters**. A set of parameters that originated from the layer's settings as a DAG node. These settings are the layers' structural configuration that cannot be changed.

ii) **SDFG parameters**. An additional set of parameters that go along with the layer's hardware building block. These parameters have an impact on the layer's performance and are the ones that the optimisation algorithm searches for during the design space exploration phase exploring the trade-offs between performance and resources.

All of the layers share a common set of SDFG parameters describing their hardware building blocks. These parameters are the number of streams accepted at the layer's input and provided at the layer's output, namely $\boldsymbol{c_n^{in}}$ and $\boldsymbol{c_n^{out}}$, the consumption rate of the layer, defined as number of elements consumed per cycle $\boldsymbol{r_n^{in}}$, and the production

rate of the layer, defined as number of elements produced per cycle $\boldsymbol{r_n^{out}}$. An "element" is defined as a single data point, like a pixel value or an individual value within a feature map, which is processed by the layer during computation. The detailed description of the parametrisation of all the layers is given below:

- **3D Convolution** - $< \boldsymbol{DAG_n^{params}}, \boldsymbol{c_n^{in}}, \boldsymbol{c_n^{out}}, \boldsymbol{r_n^{in}}, \boldsymbol{r_n^{out}}, \boldsymbol{f_n} >$
    - $\boldsymbol{f_n}$ is the vector dot product folding. Essentially is the number of multiply and accumulate (MAC) operations that take place inside the convolution in parallel.

  During the Design Space Exploration (DSE), the parameters $\boldsymbol{c_n^{in}}$, $\boldsymbol{c_n^{out}}$, and $\boldsymbol{f_n}$ are adjusted, impacting the final performance of each layer. Meanwhile the $\boldsymbol{r_n^{in}}$ and $\boldsymbol{r_n^{out}}$ depend on the $\boldsymbol{c_n^{in}}$, $\boldsymbol{c_n^{out}}$, and $\boldsymbol{f_n}$ which means they are implicitly altered during the DSE as well.

- **3D Activation** - $< \boldsymbol{DAG_n^{params}}, \boldsymbol{c_n^{in}}, \boldsymbol{c_n^{out}}, \boldsymbol{r_n^{in}}, \boldsymbol{r_n^{out}} >$
  These layers' $\boldsymbol{r_n^{in}}$ and $\boldsymbol{r_n^{out}}$ can achieve consumption/production rates of 1 without constraints from previous layers or the memory, due to the element-wise functionality of the activation operations such as Sigmoid, Swish, ReLu, and the simplicity of their operations.

- **3D Element-wise** - $< \boldsymbol{DAG_n^{params}}, \boldsymbol{c_n^{in1}}, \boldsymbol{c_n^{in2}}, \boldsymbol{c_n^{out}}, \boldsymbol{r_n^{in1}}, \boldsymbol{r_n^{in2}}, \boldsymbol{r_n^{out}} >$
  This layer's $\boldsymbol{r_n^{in1}}$, $\boldsymbol{r_n^{in2}}$ and $\boldsymbol{r_n^{out}}$ can achieve consumption/production rates of 1 without constraints from previous layers or the memory, due to their element-wise functionality and the simplicity of their operations. It should be noted that in cases when the rates in either of the inputs are restricted owing to a lower production rate of a previous layer or due to memory constraints, we equalise the layer's input rates to the lower consumption rate among them.

- **3D Global Average Pooling** - $< \boldsymbol{DAG_n^{params}}, \boldsymbol{c_n^{in}}, \boldsymbol{c_n^{out}}, \boldsymbol{r_n^{in}}, \boldsymbol{r_n^{out}} >$
  For the 3D Global Average Pooling, the consumption rate $\boldsymbol{r_n^{in}}$ is equal to 1, but the layer's production rate $\boldsymbol{r_n^{out}}$ depends on the height H, the width W, and the depth D dimensions of the input feature map. The production rate is calculated as follows: $\boldsymbol{r_n^{out}} = \frac{1}{H_n \times W_n \times D_n}$.

- **3D Batch Normalisation** - $< \boldsymbol{DAG_n^{params}}, \boldsymbol{c_n^{in}}, \boldsymbol{c_n^{out}}, \boldsymbol{r_n^{in}}, \boldsymbol{r_n^{out}} >$
  The Batch Normalisation layer normalises the input across the batch dimension,

stabilising the learning process and improving convergence speed. The parameters $\boldsymbol{r_n^{in}}$ and $\boldsymbol{r_n^{out}}$ for this layer can achieve consumption/production rates of 1, due to the normalisation being applied element-wise across each feature map.

3.3.3 SDFG Representation

Utilising the SDF theory [72], SDFG may be represented as a topology matrix $\mathbf{\Gamma}$. The nodes are represented by the columns of this matrix, while the arcs that link the nodes are represented by the rows. The data consumption/production rates for each node in each arc can be inferred by looking at the element at position $(node, arc)$ in the $\mathbf{\Gamma}$ matrix. Positive values, by convention, drive data production, whereas negative ones drive data consumption. The element $\mathbf{\Gamma}(n, a) = -1$, for example, indicates that node n consumes data at arc a at a rate of one.

The $\mathbf{\Gamma}$ matrix is decomposed into three distinct matrices, extending the SDF paradigm as proposed in fpgaConvNet [70], to enable finer control and detailed analysis of the network's dataflow properties. This approach preserves compatibility with the original topology matrix while providing a flexible representation that facilitates effective optimization and hardware mapping. The descriptions of the three distinct matrices is outlined below:

i) The stream matrix $\mathcal{S}$. This matrix contains elements that record the number of incoming and outgoing parallel streams for each node's input and output.

ii) The rate matrix $\mathbf{R}$. The rate matrix elements include the normalised data production and consumption rates of each node at each arc (defined as the number of elements produced/consumed per cycle). The values in this matrix range from 0 to 1.

iii) The data matrix $\mathcal{D}$. The width of each individual stream from the S matrix is stored the in elements of this matrix. Since all of the streams are assumed to have the same bit width, the above matrix will not be taken into consideration in this study.

The $\mathbf{\Gamma}$ matrix decomposition can be derived from the Hadamard product of the streams matrix $\mathcal{S}$, and the rate matrix $\mathbf{R}$:

$$\mathbf{\Gamma} = \mathcal{S} \times \mathbf{R} \tag{3.2}$$

The upper bi-diagonal structure of the $\mathbf{\Gamma}$ matrix, as originally described in fpgaConvNet [70], limits its ability to model branching behaviours, like in graphs where nodes have multiple outgoing or incoming arcs. To overcome this limitation, the proposed SDFG modification extends the original formulation, facilitating the creation of graphs with multiple incoming and outgoing arcs at nodes, effectively supporting branching and merging behaviours. The matrix below illustrates an example of the proposed SDFG.

$$\Gamma = \begin{array}{c} \begin{array}{ccccccc} \text{MemIn} & \text{Relu} & \text{Conv1} & \text{Swish} & \text{Conv2} & \text{Add} & \text{MemOut} \end{array} \\ \begin{pmatrix} \frac{BW}{2} & -R_{in}^{RL} & 0 & 0 & 0 & 0 & 0 \\ 0 & R_{out}^{RL} & -R_{in}^{C1} & 0 & 0 & -R_{in1}^{AD} & 0 \\ 0 & 0 & R_{out}^{C1} & -R_{in}^{SW} & 0 & 0 & 0 \\ 0 & 0 & 0 & R_{out}^{SW} & -R_{in}^{C2} & 0 & 0 \\ 0 & 0 & 0 & 0 & R_{out}^{C2} & -R_{in2}^{AD} & 0 \\ 0 & 0 & 0 & 0 & 0 & R_{out}^{AD} & \frac{BW}{2} \end{pmatrix} \end{array}$$

In the above case, BW represents the available memory bandwidth to the device off-chip memory, which is given in the device specs that are provided as input to the system along with the high level network description file. The rest of the matrix elements indicates values that have emerged from the $\mathbf{\Gamma}$ matrix calculation in Equation (3.2). Additionally, the depth of each side of a branch is computed to incorporate extra buffering for the streams that are combined at the merge points in order to ensure the correct flow of data across the design's streams as well as to equalise the rates at the merge points. These modifications not only address the limitations of the original $\mathbf{\Gamma}$ matrix but also introduce enhancements to support more complex dataflow patterns, extending the functionality of the SDFG beyond its prior description.

3.4 Design Space Exploration and Optimisation

The hardware mapping of the SDFG assumes a final streaming architecture to be inferred. Each design point in the design space has a specific combination of the involved layers' configurable compile-time parameters as they were described in section 3.3.2. By employing a set of transformations on the SDFG, the aforementioned parameters are modified while the design space is being explored utilising the simulated annealing algorithm. This heuristic optimization method is chosen for its effectiveness in navigating

large and complex design spaces. Its iterative approach, which balances exploration and exploitation through a controlled probabilistic mechanism, enables it to efficiently identify near-optimal solutions. This is particularly valuable in this study, where the design space is vast, often encompassing millions or even billions of potential design choices.

3.4.1 Model Partitioning

As previously discussed in Chapter 2.3, there are primarily two techniques for designing CNN hardware architectures.

(a) Single engine architectures rely on a more general and powerful processing unit that is time-shared across the network's layers, as well as a scheduler, to execute operations in the right sequence.

(b) Streaming architectures, such as the one presented, consist of a unique hardware block for each CNN layer that is optimised and exploits the parallelism of that layer independently of the others.

In streaming architectures, each CNN layer is assigned a unique hardware block that is optimised to exploit the parallelism of that specific layer independently from the rest of the model's layers. As the number of layers in a CNN increases or the size of the network's input grows, FPGA resource utilisation also increases. This increased resource demand can limit the parallelism achievable by each layer in order to allow all layers to fit into a single device.

Splitting network execution into smaller partitions can solve this kind of problem while at the same time exploiting the reconfiguration capabilities of the FPGA, as proposed in [24]. Furthermore, by producing a unique architecture and delivering a separate bitstream for each partition, it allows the design of more finely tuned architectures that are better tailored to each layer's characteristics. This approach also limits off-chip memory accesses to streaming data at the design's inputs and outputs, as already described in Section 2.2.3, allowing on-chip memory to be utilized for storing the model's parameters and enabling data reuse. Following this strategy, the cost of reconfiguration must be considered whenever a new partition is loaded onto the FPGA, although this cost can be mitigated by increasing the amount of data each partition has to process (i.e. by increasing the batch size), as explained in next Section 3.4.2 and Equations (3.4) to (3.6).

3.4.2 Performance and Resource Modelling

The performance and resource modelling approaches outlined below are based on the methodologies originally introduced in fpgaConvNet [70]. A more detailed discussion of performance and resource modelling can be found in Sections 6.3.1 and 6.3.2, where these approaches are further expanded and analysed on a per-layer basis.

Performance Modelling

To calculate the performance of a given design based on the topology matrix $\mathbf{\Gamma}$, an additional matrix reflecting the workload of each layer must be included. Because the topology matrix provides the throughput of each layer at its input in consumed elements/cycle and output in produced elements/cycle, constructing a matrix with the total workload of each layer, i.e. the total number of elements to be consumed and produced, allows the generation of a new matrix that provides the number of cycles each layer requires to consume its workload. More specifically a workload matrix $\mathcal{W}$ has the same structure as the topology matrix $\mathbf{\Gamma}$ in terms of nodes and arcs of the SDFG. The elements of this matrix indicate the total number of elements to be consumed at the input of each layer as well as the total number of elements to be produced at the output of each layer. By element-wise division of the $\mathcal{W}$ matrix by the $\mathbf{\Gamma}$ matrix, the final II matrix is calculated as shown below:

$$II = \mathcal{W}/\mathbf{\Gamma} \tag{3.3}$$

The II is the initiation interval matrix, and its entries represent the total number of cycles required by each layer to consume its workload completely. The maximum value of the II matrix, denoted by II_{max} determines the initiation interval of the whole SDFG of a certain partition. A simpler explanation is that the execution time (measured in cycles) of a partition is determined and constrained by the slowest node, i.e., II_{max}. The total execution time of a partition $\mathcal{P}_i$ with batch size B is given by the following equation:

$$\mathcal{L}_{\mathcal{P}_i}(\mathrm{B}, \mathbf{\Gamma}) = \frac{1}{freq} \cdot (D_{\mathcal{P}_i} + II_{max} \cdot (B - 1)) \tag{3.4}$$

where $D_{\mathcal{P}_i}$ represents the additional delays caused by pipeline depth, which corresponds to the total number of cycles required to fill the pipeline across the entire design. It is calculated by summing the individual depths of each layer along with the extra depth

introduced by additional buffering to handle branches in the design. A more detailed discussion on pipeline depth will be provided in Chapter 5.

The model's total latency is determined, as shown in Equation 3.5, by summing the latency of each individual partition, calculated in Equation 3.4, and adding the total reconfiguration time.

$$\mathcal{L}(\mathrm{G}) = \sum_{p=0}^{\mathcal{P}} \mathcal{L}_p(B, \mathbf{\Gamma}_p) + (\mathcal{P} - 1) \cdot t_{reconfig} \tag{3.5}$$

where $\mathcal{P}$ is the total number of the partitions of the model, and $t_{reconfig}$ is the reconfiguration time needed before loading each partition to the FPGA. The extra overhead caused by device reconfiguration is proportional to the number of partitions of the final solution and is independent of the batch size. Hence by increasing the number of batches processed by the model (batch size), the first term dominates the execution time and the cost of reconfiguration is amortised.

Finally the overall throughput of the proposed architecture is inferred by dividing the total workload of the model in GOPs (Giga Operations) times the batch size, by the total latency:

$$\mathcal{T}(\mathrm{G}) = \frac{Workload_G * B}{\mathcal{L}(G)} \tag{3.6}$$

Resource Modelling

The available resources of an FPGA platform can be described by a set of parameters representing the four key resource types in modern FPGAs: *DSPs*, *BRAMs*, *LUTs*, and *FFs*:

$$\mathcal{R}_{FPGA} = \{DSP, BRAM, LUT, FF\}$$

In this work, the resource utilization for a design point on the FPGA is estimated using both analytical and regression models. Of the available resource types, DSPs are utilised by *convolutional (Conv)* and *fully connected (FC)* layers. An analytical model of DSP usage for these layer types is provided below:

$$\mathcal{R}^{DSP}_{Conv} = c^{in}_n \cdot c^{out}_n \cdot f_n, \quad \mathcal{R}^{DSP}_{FC} = c^{in}_n \cdot c^{out}_n$$

BRAM utilization is also described by an analytical model, capturing the memory requirements for storing intermediate data and layer's parameters. The number of BRAM

blocks required is given by:

$$\mathcal{R}^{BRAM}(depth, words) = \left\lceil \frac{depth}{512} \right\rceil \cdot \left\lceil \frac{16 \times words}{36} \right\rceil$$

This model assumes a 16-bit fixed-point representation, with the bus width calculated as $16 \times words$. The depth and width are determined by the layer type and layer description, as described in Section 3.3.1, as well as the degree of parallelisation of the input channels (c_n^{in}), as further elaborated in Section 6.3.2. In addition to intermediate data, *Conv* and *FC* layers require extra on-chip memory for storing parameters.

For LUTs and FFs, regression models are used to estimate resource usage due to the non-deterministic nature of FPGA synthesis. These regression models are derived from data collected across numerous synthesized experiments, using various configurations of parallelisation and layer descriptions.

3.4.3 Partition Level Optimisations

As previously discussed in Section 3.3.2, each node of the SDFG has a set of fixed structural configuration parameters and a set of compile-time configurable parameters that affect its performance. Given an optimisation objective, for example throughput, the goal is to determine the best combination of these configurable parameters for each layer in the partition in order to maximise throughput. The design evaluation is carried out utilising performance and resource estimation models, which allow for the rapid traversal of the broad design space and identification of the optimum solution in minutes or hours. Alternatively, implementing, synthesising, and placing the design in an FPGA board would prevent a comprehensive study and identification of the best possible design among millions of available choices due to the lengthy duration of such complicated operations.

Configurable Parameters

Each partition layer's configurable parameters leverage its parallelism based on two factors:

- **Coarse-grain Folding:** The coarse-grain folding transformation modifies the number of parallel executions of coarse operations in each layer to achieve parallelism over the channel dimensions of the input feature-map. The primary operations of each layer can be performed simultaneously by deploying (at max) as many instances of its processing blocks as the number of channels. On Fully Connected

and 3D Convolutional layers the coarse-level parallelism can be utilised at both the input and output channels. This parallelism is achieved by updating and optimizing the compile-time parameters c_n^{in} and c_n^{out}, which are carefully selected during optimization and are incorporated into the performance and resource models. These parameters influence the stream matrix $\mathcal{S}$, which in turn affects the topology matrix $\mathbf{\Gamma}$, ultimately shaping the overall design performance. To be considered valid, a design must comply with the constraints driving this transformation.

$$c_n^{in}, c_n \in \textit{factors } C_n^{in},\ c_n^{out} \in \textit{factors } C_n^{out}$$

- **Fine-grain Folding:** The fine-grain folding transformation factor determines the parallelism of the vector dot product operation for 3D convolutional layers. This sort of parallelism specifies the number of multipliers to be used in parallel for multiplications and the number of levels on the adder trees for additions. By increasing the compile-time fine folding factor, design performance is increased at the cost of extra DSP resources. Evidently, there is a trade-off between performance and resource utilisation. For example, a completely unrolled design uses $|K_n|$ multipliers and $|K_n| - 1$ adders, yielding 1 dot product per cycle. On the other hand, restricting the setup to a single multiplier and adder yields $1/|K_n|$ dot products per cycle. $|K_n|$ represents the total number of elements in the kernel. This type of parallelism is accomplished by modifying the f_n parameter during the optimisation process. This parameter influences the rate matrix R, which in turn affects the topology matrix $\mathbf{\Gamma}$, ultimately dictating the overall design performance. The constraint on the compile-time fine-grain folding f_n is given as,

$$f_n \in \textit{factors } |\boldsymbol{K}_n|$$

Optimisation Solver

The design space exploration for a model is framed as an optimisation problem with a specific objective. This objective can be:

- **Either the maximisation of the model's throughput:**

$$\max_{G} \mathcal{T}(\mathrm{G}), \qquad \text{subject to:} \quad G \in \mathcal{C}$$

- **Or the minimisation of the model's latency:**

$$\min_{G} \mathcal{L}(G), \qquad \text{subject to:} \quad G \in \mathcal{C}$$

where $G \in \mathcal{C}$ indicates that G must belong to the set of feasible solutions $\mathcal{C}$ that satisfy all the optimisation constraints, as described in the following Section.

Simulated annealing (SA) [120], is a meta-heuristic for finding minima in non-convex functions, and it is adopted as the optimisation approach for achieving the desired objective, which in this work is *maximizing throughput*. SA was chosen as the optimization method because it is highly effective for navigating the large, complex, and non-convex design space inherent in this FPGA accelerator design problem. A simpler algorithm, like a greedy search, would quickly get trapped in a suboptimal solution or would require very carefull tuning and a great amount of time, while an exhaustive search is computationally impossible. The implementation of SA for this particular optimisation problem is given in Algorithm 1.

Algorithm 1 Simulated Annealing Optimisation Algorithm

1: $\tau \leftarrow \tau_{start}$, $G_{new} \leftarrow G_{init}$ ▷ initialisation
2: **while** $\tau > \tau_{min}$ **do**
3: $G_{prev} \leftarrow G_{new}$ ▷ store previous design
4: $\mathcal{O}_{prev} \leftarrow \mathcal{O}_{total}(G_{prev})$ ▷ get design performance
5: $G_{new} \leftarrow$ random transformations on G_{prev}
6: $\mathcal{O}_{new} \leftarrow \mathcal{O}_{total}(G_{new})$ ▷ get new design performance
7: **if** $G_{new} \in \mathcal{C}$ **then**
8: **if** $\mathcal{O}_{new} < \mathcal{O}_{prev}$ **then**
9: **if** $\psi(\mathcal{O}_{prev}, \mathcal{O}_{new}, \tau) < x \sim U(0,1)$ **then**
10: $G_{new} \leftarrow G_{prev}$ ▷ reject new design
11: $\tau = \kappa \cdot \tau$ ▷ reduce temperature

where,

$$\psi(O_{prev}, O_{new}, \tau) = \exp\left(\frac{O_{new} - O_{prev}}{\tau}\right)$$

The main objective of the algorithm is to seek the global minimum or maximum of a given cost function; in this case, the cost function is the maximisation of the throughput, which is derived from Equation (3.6). Starting at an initial state by producing random values for the parameters of the computation nodes, transformations to the hardware graph G are applied iteratively generating new states that may be accepted or rejected based on a policy described in Algorithm 1. The SA hyperparameters are configured as,

$\tau_{start} = 10$, $\tau_{min} = 1 \times 10^{-6}$, and the cooling rate $\kappa = 0.99$, while a warm start is executed prior to the execution of the optimiser. The system's optimisation variables include the parameter space, such as c_n^{in}, c_n^{out}, f_n, of all the instantiated building blocks of the current state of the G. Each new state is evaluated based on the system's predetermined constraints, as described in the Section below, and is only accepted if it satisfies all of them.

Optimisation Constraints

To ensure the validity of the generated designs during the design space exploration process, some constraints must be met as regards the transformations and the alternations of the SDFG, G, of the model. These constraints are categorised below:

- **Resource Constraints:** These constraints ensure that the FPGA implementation does not exceed available hardware resources (DSP, BRAM, LUT, FF). They are defined by the following equations:

$$\mathcal{R}(\mathcal{P}_i) = \sum_{n_j \in \mathcal{P}_i} \rho(n_j | c_j^{in}, c_j^{out}, f_j)$$

$$\max\{\mathcal{R}(\mathcal{P}_i) : \mathcal{P}_i \in \mathcal{P}\} \leq \mathcal{R}_{FPGA}$$

  Here, $\mathcal{R}(\mathcal{P}_i)$ represents the total resources used by partition $\mathcal{P}_i$, calculated by summing the resource usage ρ of each node n_j in the partition, derived from the resource modelling. This usage depends on input channels c_j^{in}, output channels c_j^{out}, and fine-grained parallelism f_j. The second inequality ensures that the maximum resource usage across all partitions does not exceed the available FPGA resources $\mathcal{R}_{FPGA}$.

- **Parameters Constraints:** These constraints ensure proper parallelisation of the individual operations and are divided into coarse and fine operations.

  - *Coarse Operations:* For coarse operations, the total number of input and output channels (C_n^{in} and C_n^{out}) must be divisible by their respective parallelism factors (c_n^{in} and c_n^{out}).

$$C_n^{in} \mod c_n^{in} = 0 \quad \forall n \in G$$

$$C_n^{out} \mod c_n^{out} = 0 \quad \forall n \in G$$

For certain layers with dependent input and output channels, such as activation layers, element-wise layers, pooling layers, the input and output channel folding must be the same creating an extra and specific constraint for these layers:

$$c_n^{in} = c_n^{out} \quad \forall n \in G'$$

where G' is a subset of the computation nodes of G that comprise of the aforementioned layers.

– *Fine Operations:* The fine operations constraint ensures that the fine-grained parallelism f_n is a factor of the layer's kernel size, and it is applicable to layers like Convolutional and Fully Connected layers.

$$f_n \mod |K_n| = 0 \quad \forall n \in G$$

where $|K_n|$ is the kernel size in terms of number of elements.

- **Bandwidth Constraints:** This constraint ensures that the total data transfer between the FPGA and external memory (usually DRAM) doesn't exceed the available bandwidth of the memory interface. They can be expressed as:

$$\mathbf{BW}^r(\mathcal{P}_i) = \sum_{n_j \in \mathcal{P}_i} R_{n_j} \implies n_j \in G_{input}$$

$$\mathbf{BW}^w(\mathcal{P}_i) = \sum_{n_j \in \mathcal{P}_i} R_{n_j} \implies n_j \in G_{output}$$

$$\mathbf{BW}^r(\mathcal{P}_i) < \mathbf{BW}^r_{FPGA}, \quad \mathbf{BW}^w(\mathcal{P}_i) < \mathbf{BW}^w_{FPGA} \quad \forall \quad \mathcal{P}_i \in \mathcal{P}$$

The initial two equations define the bandwidth requirements for a specific partition $\mathcal{P}_i$. They focus on two key sets of nodes:

– G_{input}: This set includes the nodes that read data from external memory.

– G_{output}: This set includes the nodes that write data to external memory.

For each partition $\mathcal{P}i$, we calculate its total bandwidth by summing the rates of all nodes within it that belong to either G_{input} or G_{output}. These rates are specified in the rate matrix $\mathbf{R}$ described in Section 3.3.3. The resulting sums represent the total bandwidth needed for reading from and writing to off-chip memory, respectively, for that particular partition $\mathcal{P}_i$.

In the third equation, $\mathbf{BW}^{r/w}(\mathcal{P}_i)$ is the total read and write memory bandwidth for a partition $\mathcal{P}_i$, and $\mathbf{BW}^{r/w}_{FPGA}$ is the available memory bandwidth of the given platform.

In essence, the set of these equations calculate the total bandwidth needed for a particular partition (P_i) and ensures they don't exceed the maximum available bandwidth of the given platform.

By adhering to these constraints, we can generate valid designs that efficiently manage data transfer, appropriately parallelise operations, and effectively utilise FPGA resources, thereby optimising the overall performance of the neural network model implementation on FPGAs.

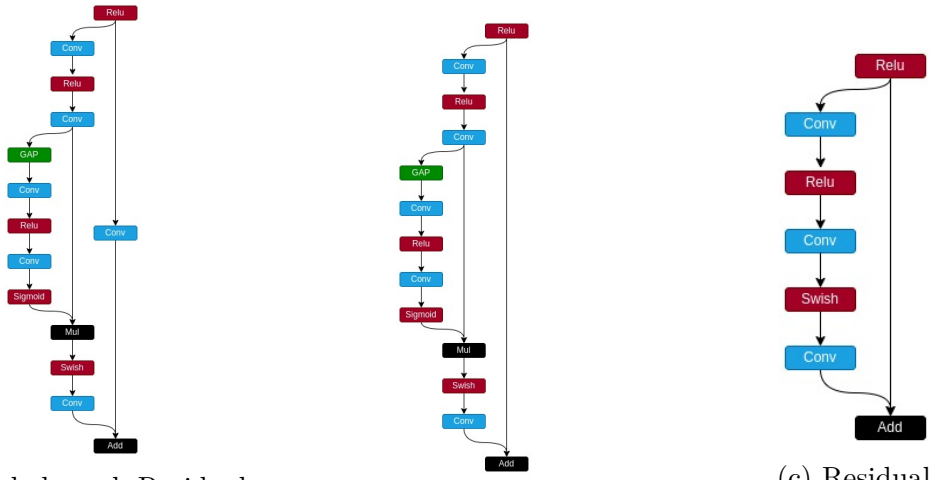
3.4.4 Model Specific Optimisations

X3D Partitioning

In this work, the partitions are manually defined based on three main block types, identified and tailored to the architecture and specific characteristics of the X3D model. Figure 3.4 depicts the structure of the defined block types. The first block type in Figure 3.4a consists of a double branch structure with a convolution layer at the outer branch's skip connections. The second block in Figure 3.4b is the same as the first but without the convolution layers at the skip connection, while the third block in Figure 3.4c follows the architecture of a standard residual block. The X3D model comprises 26 such blocks, 4 of which are of type *'Double-branch Residual with Conv'* (Db Res Conv), 11 of which are of type *'Double-branch Residual'* (Db Res), and 11 of which are of type *"Residual"* (Res). Each block type, as can be observed from their structure, displays branching behaviour, with the first two block types exhibiting double branching owing to the squeeze and excitation module [48].

GAP Optimisation

During the development of the proposed design, the squeeze and excitation modules were found to significantly hinder throughput. This bottleneck primarily stems from the 3D Global Average Pooling (GAP) layer, which requires averaging over the entire feature map. This operation introduces delays in the design pipeline and necessitates increased branch stream buffering. The performance limitation arises from two main factors:



(a) Double-branch Residual with Conv

(b) Double-branch Residual

(c) Residual

Fig. 3.4 The basic X3D model block types. We identified and defined these block types based on their structure and receptiveness inside the X3D model architecture.

- Interdependence between layers: The output of the GAP layer is needed for the execution of subsequent layers, which in the case of X3D, is always a pointwise convolution. This operation requires the all input channels to be present to start producing its output, resulting in a dependency that causes pipeline stalling.
- On-chip memory constraints: The FPGA's restricted on-chip memory resources get constrained by the larger buffer sizes associated with this operation.

To address these challenges, we introduced an innovative approximation technique, as illustrated in Figure 3.5. The main points of this approach are described below:

- Result Reuse: The GAP layer re-uses the results of its previous execution, which are saved in the on-chip memory, for each subsequent volume of data (i.e., every subsequent batch).
- Immediate Forwarding: These saved results are forwarded immediately to the next layer in the pipeline, eliminating the need to wait for the current batch's GAP calculation.
- Asynchronous Update: While the pipeline continues uninterrupted, the GAP layer calculates and saves the current batch's results to the on-chip memory. This update occurs in parallel with the processing of subsequent layers, ensuring it doesn't affect the pipeline's other stages.

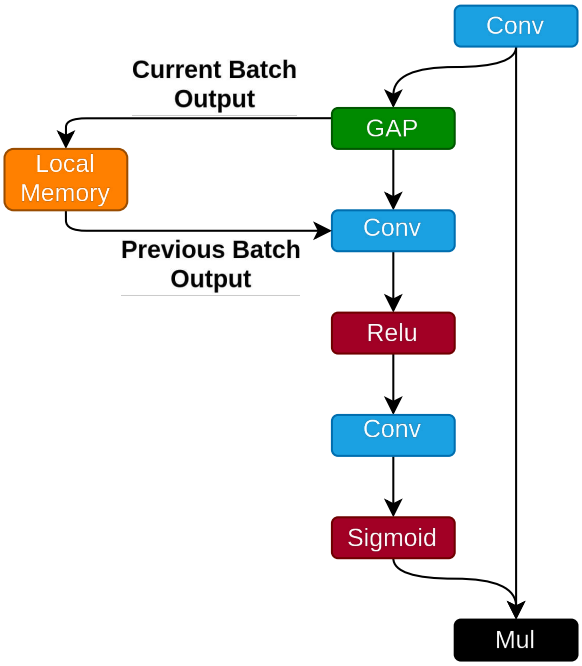


Fig. 3.5 Illustration of the proposed streaming-centric approximation on GAP layers for improving the throughput of X3D.

- Reduced Buffering: This approach significantly reduces the need for extensive buffering, as the pipeline no longer needs to wait for the GAP calculation to complete before proceeding.
- Improved Throughput: By eliminating pipeline stalls and reducing buffer requirements, this optimisation substantially improves overall throughput.

The trade-off in this approach is a reduction in accuracy due to using the previous batch's GAP results. However, when evaluated on the UCF101 benchmark (Section 3.5.2), this approximation had negligible impact on the model's accuracy, which is even more apparent in streaming scenarios where consecutive batches typically exhibit similar statistical properties.

3.5 Performance Evaluation and Results

3.5.1 Experimental Setup

For evaluation purposes, the AMD/Xilinx ZCU102 board, featuring the Zynq UltraScale+ MPSoC XCZU9EG device, has been used as the target platform. In all of the experiments

the target frequency was set to 140 MHz. Vitis HLS and Vivado Design Suite (2021.2) were used, while the reported resource results are after place and route. In all the designs the arithmetic precision used was 16-bit fixed point arithmetic as described in Section 3.5.2. The UCF101 [1] HAR benchmark was used to assess the correctness of the X3D model and position it in relation to prior studies. In addition to the baseline architecture, a second design is proposed and evaluated, utilising the statistics from previous batches of GAP layers as described in Section 3.4.4.

Since the authors of the X3D [6] have not published the model's performance in that specific benchmark, we have trained/fine-tuned the X3D model and then evaluated it on the UCF101 benchmark. Following the model's fine-tuning, the evaluation was carried out utilising the widely adopted strategy used on HAR benchmarks, which results in a total of 30 predictions that are averaged from a uniform sampling of 10 temporal clips C_t for each video and spatial sampling of 3 crops in each frame, resulting in 3 spatial clips C_s. Furthermore, to be equivalent and directly comparable to previous studies, the impact of employing fixed-point precision has been investigated, by converting the model weights and all intermediate feature mappings from 32-bit floating point to 16-bit fixed-point arithmetic.

3.5.2 Quantisation Experiments

Figure 3.6 depicts the results of quantisation of both the weights and feature maps over different combinations of word lengths under fixed point number representation. The experiments were conducted on the first split of UCF101 with a single clip evaluation (compared to 30 clip evaluation as described above). The accuracy achieved for this split with floating point (Float32) feature maps and weights is 95.71%. On a 16-bit fixed point precision for both the weights and feature maps (W16A16) we observe a drop in accuracy of 0.28pp. Keeping the feature maps at 16 bits and moving weights to 15-bit precision (W15A16) the accuracy drops by 0.66pp, at 14-bits weights (W14A16) the accuracy drop is equal to 1.19pp, and for 13 bits (W13A16) the drop in accuracy is at 3.23pp. In Figure 3.6 all the possible combinations of weights and feature maps precisions are presented. An interesting insight emerged from this evaluation is that the feature maps are less sensitive to reduction of the word length used to represent them compared to the weights of the model. For all the experiments we have explored and kept the best combination of the number of integer and fractional bits for each word length both for weights and feature maps. Finally we have evaluated the model on all three splits of UCF101 with a 30 clip evaluation using 16-bit fixed point for weights ($Q6.10$) and for

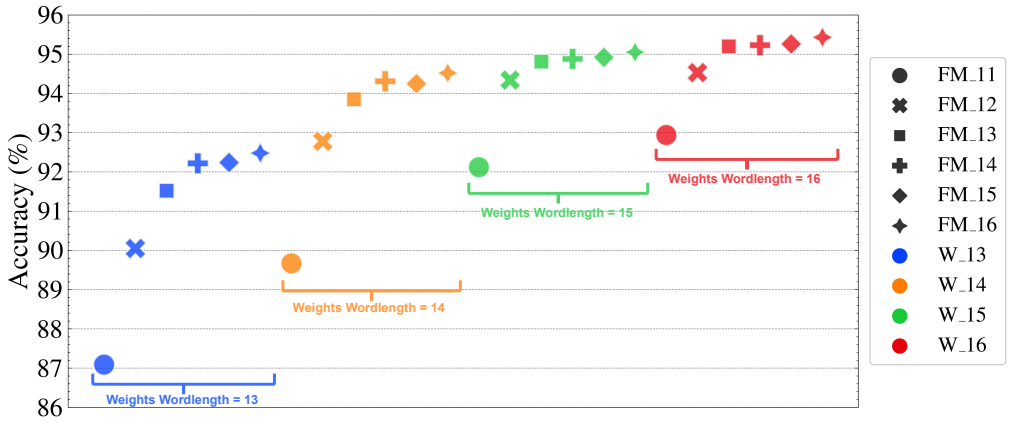


Fig. 3.6 Quantisation results on UCF101 over different word lengths. Wx denotes the word length of the fixed point representation of the weights, while FMx for the feature maps

feature maps ($Q7.9$). It is observed that the accuracy drop was negligible dropping from 96.56% to 96.43%.

These results demonstrate that fixed-point quantization can achieve comparable accuracy to floating-point precision with careful selection of word lengths, offering significant potential for reducing resource usage without compromising model performance.

3.5.3 Performance Evaluation

In this section (and chapter), the performance evaluation focuses specifically on the mapping of the X3D network to FPGA, as a case study to demonstrate the proposed methodology. While the results provide valuable insights into the effectiveness of the flow presented in Figure 2.13, no claims are made about the generalization of these findings to other 3D CNN models. A more comprehensive evaluation of the proposed flow, including its application to a broader range of 3D CNN architectures, will be presented in the next chapter to assess its adaptability and robustness across different network designs.

Accuracy of Modelling

To ensure that the performance of X3D predicted by the modelling is representative of the actual performance provided by co-simulation results, a series of experiments has been conducted for each supported layer type with different configurations for the layer input and output shapes, as well as the coarse in and coarse out factors. The metric of Mean Percentage Error (MPE) was used to measure the error on the predicted

vs co-simulation throughput for all the experiments. It was observed that the model consistently under-predicts the throughput, providing a conservative but safe estimate for the Design Space Exploration. As can be seen from Figure 3.7, the errors vary depending on the coarse in and coarse out factors; the bigger the coarse factors get the smaller the error becomes. On average, layer type 1 has an MPE of 11.92%, layer type 2 has 17.32%, and layer type 3 has 5.03%.

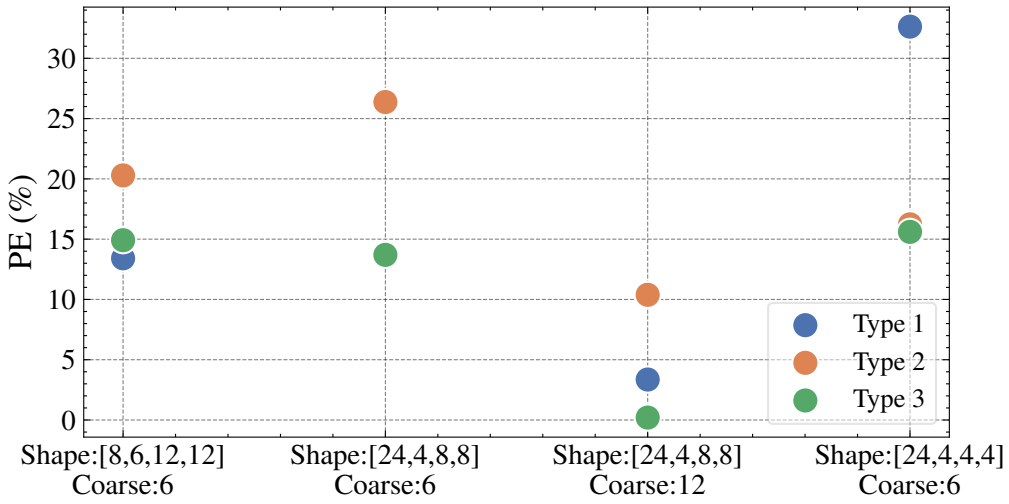


Fig. 3.7 PE on throughput (clips/s) between modelling and co-simulation results.

Experimental Results

This subsection presents the experimental results, showcasing the performance of the FPGA-based X3D implementation. It compares the results with other state-of-the-art models and hardware platforms, highlighting improvements in key metrics such as throughput, accuracy, and resource efficiency. A batch size 100 was chosen to amortise reconfiguration cost. These batch size values can be used in systems that analyse multiple videos in parallel, or in evaluation processes that use multiple clips from a single video, as already described in detail in Section 2.3.3.

To the best of our knowledge, no previous works have targeted the acceleration of X3D onto FPGA as of the date of authoring this study. To position within the previous works, and because most current works focus on the C3D model targeting different FPGA platforms, the assessment is undertaken on a number of different metrics: **Clips per second (Clips/s)**. This metric is the total number of clips processed over a second. The values on this metric on Table 3.3 are measured and reported for a batch

size of 100 as denoted in the Table's footnotes. The positioning of the proposed designs related to the existing works can be seen in Figure 3.8. The proposed architecture is positioned in the Pareto front of accuracy over clips/s compared to the existing works in the field.

Throughput (GOp/s). To be fully aligned with the results of the existing works, GOp/s are accounted as the MAC (multiply and accumulate) operations per second. For batch size 100, the design generated by the proposed method achieves better throughput compared to [109, 110, 116] while at the same time achieving better accuracy among all the existing works.

DSP efficiency (GOp/s/DSP). This metric is the normalised throughput over the DSPs used. In this work, the DPSs used are averaged over all the 26 pre-defined partitions. A positioning of the designs proposed here in relation to the existing works for a favourable batch size of 100, showcases a comparable DSP efficiency to most of the works.

Performance Density (Op/s/DSP/cycle). This metric is the normalized throughput over the DSPs used and the clock frequency. In the proposed method the DSPs used are averaged over all the 26 defined partitions. A comparison with the existing works can be found on Table 3.3, where the proposed solution achieves comparable performance density ranging from 0.89× to 0.15×.

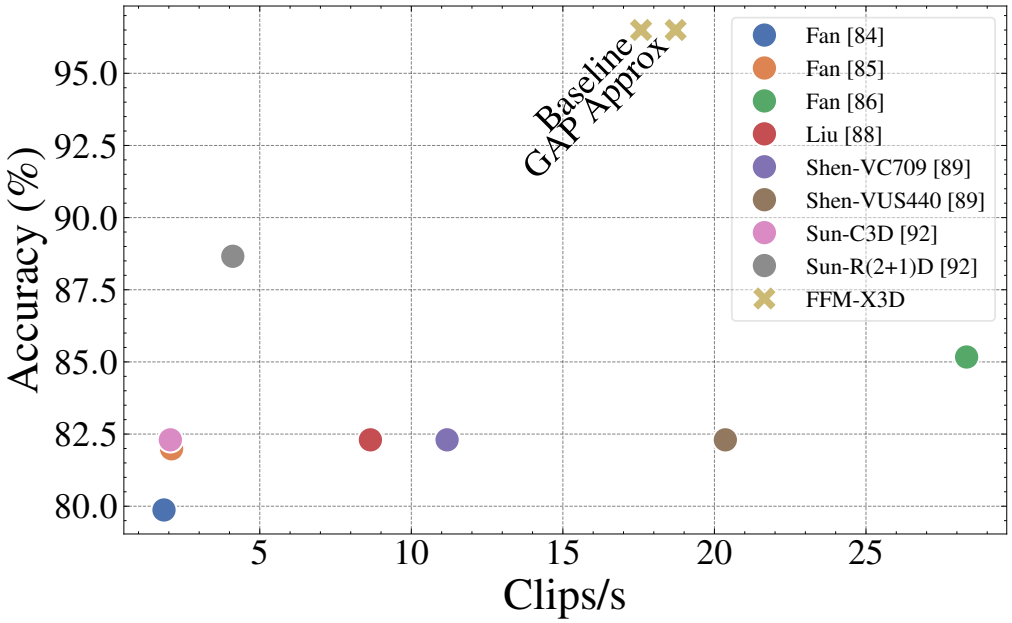


Fig. 3.8 Positioning of FFM-X3D on the Pareto Front of Accuracy vs. Clips per Second.

Given that the proposed design focuses on throughput, the related works are also evaluated based on this metric. In comparison to M. Sun et al. [116], the sole effort targeting the same FPGA platform, the designs generated by the proposed method outperform their C3D and R(2+1)D counterparts by 1.52× and 1.07×, respectively, in terms of throughput expressed as GOp/s. Table 3.3 summarises the positioning of this chapters designs in relation to the rest of the existing works.

Table 3.2 shows a performance comparison of the proposed (GAP approximation) design on a ZCU102 FPGA board versus a mid-range Nvidia GPU platform. A GeForce RTX 2060 with 1920 CUDA cores is the target GPU utilising CUDA and CuDNN libraries to optimise its solution. However, due to the memory limitations of the GPU platform, we were restricted to using a batch size of 20 for evaluation. Under this constraint, our FPGA-based design achieves a power efficiency of 0.17 Clips/s/Watt, compared to the GPU's 0.66 Clips/s/Watt. Notably, the performance of the proposed FPGA design is constrained by the smaller batch size. For larger batch sizes, such as 100, it could achieve up to 0.72 Clips/s/Watt, which is more than four times the efficiency observed under the current batch size limitation. However, the performance of the specific GPU under these conditions remains unknown, as it cannot be evaluated for larger batch sizes.

Table 3.2 Performance comparison versus GPU

	GPU	FFM-X3D (Ours)
Platform	GeForce RTX 2060	ZCU102
Frequency	1.2 GHz	140 MHz
Memory Capacity†	6 GB	4 GB (44.1 Mb)
Memory Bandwidth	336 GB/s	19.2 GB/s
Power (W)*	79.08	26
Batch Size	20	20
Precision	Float32	W16A16
Clips/s (cps)	52.2	4.42
Clips/s/Watt (cps/W)	0.66	0.17

* The GPU power estimates obtained from the *nvidia-smi* tool, while the FPGA power estimates derived from the Vivado tool estimations.
† For the ZCU102 the number in the parenthesis indicates the on-chip memory capacity.

Table 3.3 Positioning of FFM-X3D in the space of 3D CNN HAR models

	H. Fan [109]	H. Fan [110]	H. Fan [111]	Z. Liu [113]	J. Shen [2]†		M. Sun [116]		FFM-X3D (Ours)	
					VC709	VUS440	C3D	R(2+1)D-18	Baseline	GAP-approx
Model	C3D	C3D	E3D	C3D	C3D	C3D	C3D	R(2+1)D-18	X3D-M	X3D-M
GMACs	38.61	38.61	6.1	38.61	-	-	38.61	26.14	6.97	6.97
Precision	W16A16	BFP24	Float32	W16A16	W16A16	W16A16	W16A16	W16A16	W16A16	W16A16
Accuracy	79.87 %	81.99 %	85.17 %	82.3 %	82.3 %	82.3 %	82.3 %	88.66 %	96.5 %	96.5 %
Board	ZC706	ZC706	Intel SX660	VC709	VC709	VUS440	ZCU102	ZCU102	ZCU102	ZCU102
DSP %	90 %	86.6 %	93.3 %	99.8 %	42 %	53 %	48 %	48 %	86 %	85 %
Frequency (MHz)	172	200	150	120	150	200	150	150	140	140
Throughput (clips/s) *	1.84	2.09	28.32	8.65	11.18	20.36	2.05	4.11	17.58	18.72
GOp/s *	70.41	80.12	172.8	330.74	427.29	778	78.44	111.71	112.41	119.83
GOp/s/DSP *	0.087	0.103	0.109	0.092	0.281	0.511	0.065	0.092	0.052	0.055
Op/DSP/cycle *	0.511	0.519	0.727	0.774	1.874	2.559	0.435	0.613	0.367	0.391

* Favourable batch size 100. † The C3D model used is different/smaller version from the original one [3].

3.6 Conclusions

This chapter has introduced a novel approach for optimising and mapping 3D Convolutional Neural Networks (CNNs)—with a specific focus on the X3D model—onto Field-Programmable Gate Array (FPGA) platforms. The work addresses the growing need for efficient hardware acceleration of complex deep learning models in video-related applications. Our contributions leverage the computational efficiency and adaptability of FPGAs to address the unique challenges posed by 3D CNN architectures. The following contributions have been made toward FPGA-based mapping and optimisation of the X3D model:

- Extension of the Synchronous Data Flow (SDF) graph model to better capture performance requirements unique to 3D CNN mapping on streaming architectures by explicitly addressing irregular block structures with branching which are common in modern CNN models.
- An automated process was developed to optimise X3D, a state-of-the-art HAR model, targeting the ZCU102 FPGA board, factoring in device-specific constraints such as available resources and memory bandwidth.
- Implementation of model-specific optimisations, such as the innovative approximation technique for Global Average Pooling (GAP) layers, which significantly improved throughput by reducing pipeline stalls and buffer requirements.
- Comprehensive evaluation of the proposed architecture (FFM-X3D) against existing works, demonstrated competitive performance in terms of throughput, accuracy, and resource efficiency. Notably, the proposed designs achieved 17.58 clips/s (baseline) and 18.72 clips/s (with GAP approximation) while maintaining 96.5% accuracy on the UCF101 benchmark.
- Successful quantisation of the X3D model to 16-bit fixed-point precision with minimal accuracy loss (0.13% on UCF101), enabled efficient FPGA implementation.

The proposed approach pushes the current performance-accuracy Pareto front for FPGA-based 3D CNN acceleration, enabling the deployment of sophisticated architectures like X3D in resource-constrained environments.

While the proposed approach demonstrates notable improvements in performance and efficiency for FPGA-based acceleration of the X3D model, several limitations and

challenges remain. One primary limitation is the need for extensive manual tuning and partitioning of the model. This manual partitioning is highly model-specific and tailored to optimize resource allocation and computational flow, which does improve results but comes with significant time costs. The process is notoriously labor-intensive and can be challenging to replicate or adapt for different devices or architectures, thereby limiting scalability and adaptability across various FPGA platforms.

To overcome these limitations, in the next chapter, we will explore automated and adaptive partitioning strategies. Automating the partitioning process could make the optimisation process less time-intensive and more easily transferable across different FPGA devices and model architectures. Additionally, improving the accuracy of performance and resource prediction models is essential for streamlining the design space exploration process. Integrating more accurate prediction models would enable precise early-stage assessments of resource and performance requirements, thereby improving overall design efficiency. The next chapter builds on this foundation by introducing throughput-oriented optimization techniques, with a focus on automating and streamlining the model mapping process, focusing mainly on the partitioning strategies. These techniques aim to support high-throughput 3D CNN applications, advancing the scalability and adaptability of FPGA-based solutions across diverse and complex architectures.

Chapter 4

Charting the Course: Streamlining FPGA-Based Optimisation for 3D CNNs Targeting Throughput-Oriented Designs

4.1 Introduction

This chapter introduces fpgaHART[1] (fpga-based Human Action Recognition Toolflow), a throughput-oriented toolflow developed to optimise and map 3D Convolutional Neural Networks (CNNs) onto FPGA devices. 3D CNNs are widely utilised in fields like surveillance, autonomous systems, human behaviour monitoring, and video retrieval, where capturing temporal or depth-related features is essential. Despite their effectiveness, these networks pose significant computational and memory demands, particularly challenging for resource-constrained FPGA platforms. To address these challenges, fpgaHART leverages the flexibility and parallelism of FPGAs to achieve high-throughput, efficient designs suitable for real-time, low-power applications.

Building on the limitations identified in the previous chapter, this chapter addresses bottlenecks associated with manual partitioning and model-specific tuning. fpgaHART introduces an automated and adaptive model partitioning process, aiming to make the optimisation of 3D CNNs both scalable across devices and adaptable to diverse model architectures. The toolflow employs an automated partitioning strategy, which identifies

[1] https://github.com/ptoupas/fpgahart

optimal reconfiguration points that, alongside the per-partition optimisation strategy, as described in the previous chapter, and the refined performance and resource prediction models, enable fpgaHART to handle complex architectures while exploring the design space in an automated way, producing throughput-optimised FPGA designs.

The key contributions of this chapter are as follows:

- **Development of the fpgaHART Toolflow**: fpgaHART is designed as a throughput-oriented optimisation and mapping toolflow for 3D CNNs on FPGA devices, transitioning from a model- and device-specific approach, introduced in the previous chapter, to a more generic framework that supports mapping multiple models to multiple FPGA devices. This adaptability allows fpgaHART to serve a broader range of 3D CNN architectures, extending its use to applications beyond HAR.

- **Automated Model Partitioning**: A key feature of fpgaHART is its automated partitioning strategy, which identifies optimal reconfiguration points even when partitioning is required mid-branch. This automation reduces the time-intensive nature of manual partitioning and facilitates the identification of optimal partitions, enabling efficient optimisation across diverse hardware setups and complex 3D CNN models with challenging branching structures.

By automating the optimisation and mapping process, fpgaHART not only reduces the manual effort required for partitioning but also expands the design space, enabling the exploration of a broader range of design choices, allowing for the identification of better design points. While the toolflow is motivated by the specific needs of HAR, it is adaptable to other 3D CNN architectures, expanding its potential applications for efficient 3D CNN processing across various domains.

In Table 4.1 below we provide a comparative overview of the 3D CNN models supported by fpgaHART and other related works, demonstrating fpgaHART's compatibility across a range of 3D CNN architectures. Unlike previous works, which often focused on a limited set of models tailored to specific hardware configurations or application types, fpgaHART is designed to accommodate a broader range of architectures. This versatility is achieved through key contributions, including the expansion of the design space to explore optimal reconfiguration points when partitioning the model's computational graph and the extension of performance modelling to handle multi-input/multi-output configurations. The table underscores the adaptability of fpgaHART, demonstrating its potential as a versatile solution for FPGA-based 3D CNN acceleration.

Table 4.1 Comparison of 3D CNN model support in fpgaHART versus related works, highlighting the range and diversity of supported architectures.

3D CNNs	Fan et al. [109]	Fan et al. [110]	Liu et al. [113]	Sun et al. [116]	Shen et al. [2]	fpgaHART (Ours)
C3D [3] (2013)	✓	✓	✓	✓	✓	✓
R(2+1)D-18 [97] (2018)	✗	✗	✗	✓	✗	✓
R(2+1)D-34 [97] (2018)	✗	✗	✗	✗	✗	✓
SlowOnly [42] (2019)	✗	✗	✗	✗	✗	✓
X3D [6] (2020)	✗	✗	✗	✗	✗	✓

4.2 Background

This section introduces a range of 3D CNN models that we use to demonstrate and evaluate the fpgaHART toolflow. As in Chapter 3, we provide a detailed overview of these models, emphasizing their varied workloads, architectures, memory demands, and the different operations and layers that define them. The C3D model, was introduced by Tran et al. [39] in 2014, and was one of the pioneering approaches to learning spatiotemporal features using deep 3D convolutional networks. The model was trained on a large-scale supervised dataset, demonstrating considerable advancements in video analysis relative to prior efforts by concurrently integrating spatial and temporal information utilising 3D convolutions. The C3D model features a simple linear architecture comprising 27 sequential layers, including 8 convolutional layers, each employing a $3 \times 3 \times 3$ convolution kernel. This model established the foundation for subsequent advancements in 3D CNNs, illustrating the effectiveness of 3D convolutions on video-related tasks.

In 2018, Tran et al. [97] introduced the R(2+1)D architecture, which extends the advancements of 3D CNNs by incorporating them into the residual learning framework. This model demonstrates the higher accuracy of 3D CNNs compared to 2D CNNs on video-related tasks. They proposed an approach to reduce the computational complexity of 3D convolutions by decomposing the 3D convolutional filters into two separate operations: a 2D spatial convolution followed by a 1D temporal convolution. This decomposition led to a more efficient spatiotemporal convolutional block, known as "R(2+1)D". The integration of these blocks into a ResNet-like architecture enabled the R(2+1)D model to achieve

state-of-the-art performance on benchmarks such as Kinetics [103] and Sports-1M [121], while also considerably decreasing computational costs relative to conventional 3D CNNs.

In 2019, Feichtenhofer et al. [42] introduced the SlowFast network, a novel two-pathway architecture for video recognition with a slow pathway capturing spatial semantics at a low frame rate and a fast pathway focusing on rapid motion at a high frame rate. The two pathways are connected by lateral links, enabling them to share information and complement each other. This dual-rate design effectively balances slow-changing spatial features with fast-evolving motion, resulting in strong performance on video recognition tasks like action classification and detection, and achieving state-of-the-art accuracy on benchmarks such as Kinetics [103], Charades [107], and AVA [108]. In our work, we focused on the SlowOnly variant of the SlowFast model, which uses only the Slow pathway to capture spatial semantics at a low frame rate, as object identity and appearance change more slowly. After analysing the trade-offs, we found that the increased complexity, due to the bilateral connections in the full SlowFast model, along with the higher computational workload (66.1 vs. 54.75 GFLOPs) and parameter count (34.6M vs. 32.45M), was not justified by the marginal accuracy gain (76.65% vs. 76.45% Top-1 accuracy on Kinetics-400).

Table 4.2 3D CNN models characteristics

	C3D	Slowonly	R(2+1)D-18	R(2+1)D-34	X3D
FLOPs (G)[†]	38.61	54.81	8.52	12.91	6.97
Parameters (M)	78.41	32.51	33.41	63.72	3.82
Num. of Layers	27	174	82	154	396
Num. of Conv Layers	8	53	37	69	115
Spatial dimensions	112×112	256×256	112×112	112×112	256×256
Num. of Frames	16	8	16	16	16
UCF101 Accuracy (%)	83.2	94.54	88.66	92.27	96.52

[†] FLOPs are reported as MAC operations.

The selection of these models highlights their significance in the progression of 3D CNNs, as each of them introduced essential concepts from the 2D domain into the 3D domain. For example, the SlowOnly model adapted the popular ResNet [9] architecture style into the 3D domain, while X3D integrated depthwise separable convolutions [44] and Squeeze-and-Excitation [48] blocks into the 3D domain, resulting in efficient 3D CNN architectures similar to MobileNets [38] in the 2D domain. Similarly, the R(2+1)D model introduced 3D-specific optimizations, such as the (2+1)D modules, reducing the computational cost of 3D CNNs, without compromising their performance. After 2020,

the field began shifting towards Vision Transformer (ViT)-based models, as discussed in Section 2.3.2 and illustrated in Figure 2.14. However, these models have significant computational requirements, making them less practical for the scope of this study, which emphasizes more efficient workloads fitting constrained FPGAs. Consequently, we chose to concentrate on these pivotal 3D CNN architectures, which remain relevant and impactful for applications targeting edge devices and embedded systems.

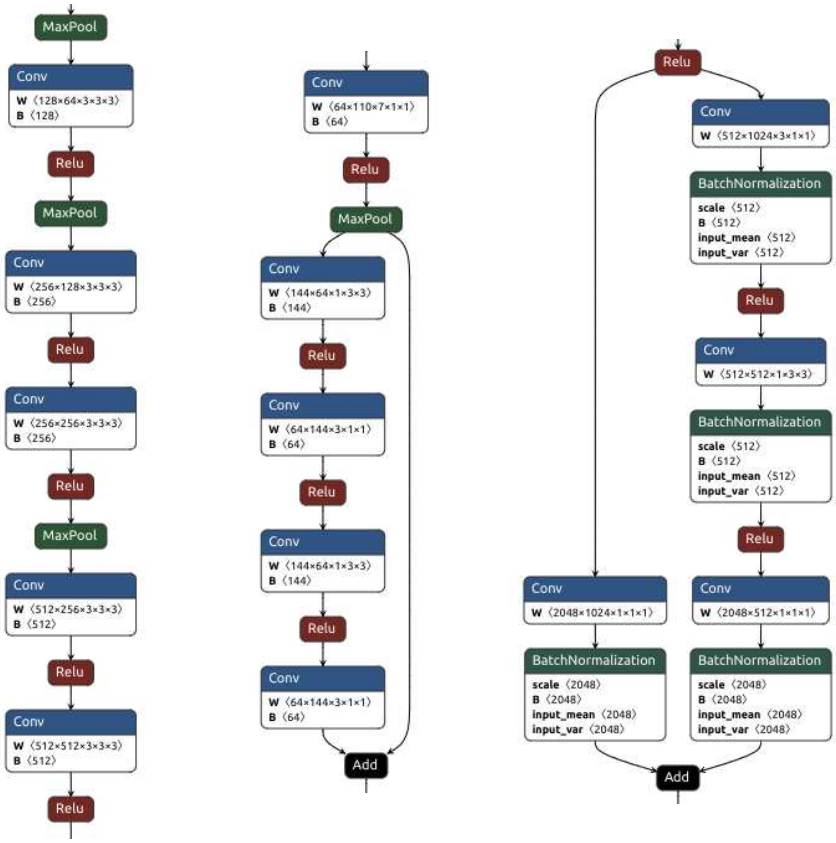


Fig. 4.1 Architecture variations across three 3D CNN architectures-C3D, R(2+1)D, and SlowOnly (left to right), showcasing the different layer connectivity and operation types used in each model. The notation $W < F \times C \times D \times H \times W$ in 3D convolutional layers represents the convolutional kernel where F is the number of filters, C is the number of channels, while D, H, W are the depth, height, and width dimensions of the kernel.

Table 4.2 presents a detailed breakdown of the previously discussed models, highlighting their specific characteristics. It includes information on workload (measured in MACs), the number of parameters, performance metrics such as accuracy, and layer-related details. This comparison helps illustrate the differences between the models in terms of computational and memory demands, architecture, and overall efficiency.

Furthermore Figure 4.1 highlights the diverse architectural workflows in 3D CNN models, showcasing variations in layer connectivity, long-range dependencies, and operation types.

4.3 fpgaHART Toolflow

The fpgaHART toolflow is designed to automate and optimize the deployment of 3D CNN models on FPGA platforms, addressing the unique challenges associated with reconfigurable hardware acceleration. The toolflow accepts inputs from popular deep learning frameworks such as PyTorch [50] and TensorFlow [51], after being converted into the ONNX format[2]. Alongside the ONNX model, the toolflow also takes FPGA platform specifications and application performance requirements as additional inputs. The FPGA specifications provide essential details about the hardware constraints and resources available, while the performance requirements help guide the optimization process towards maximizing throughput. Upon processing these inputs, fpgaHART generates an FPGA bitstream tailored to the characteristics of both the model and the hardware.

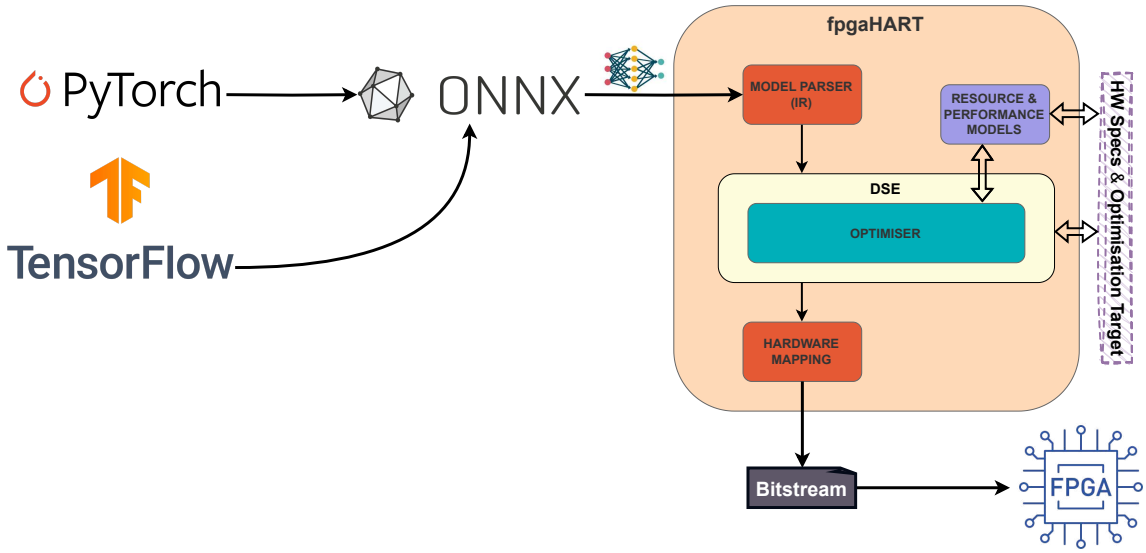


Fig. 4.2 High level overview of the fpgaHART toolflow architecture. The three distinct but connected parts are being shown in the diagram. The entry point which is the model parser creates the intermediate representation (IR) and feeds that to the optimiser. During the DSE, the optimiser exchange information with the resource and performance models. Once the optimisation is finished the automated hardware mapping tool converts IR into HDL code and then into a bitstream.

[2]https://onnx.ai/

The core of the toolflow consists of three interconnected components: the Model Parser, the Automated Design Space Exploration (DSE) Optimiser, and the Network Hardware Mapping, as shown in Figure 4.2. These components work together to transform high-level model descriptions into FPGA-specific implementations that meet application performance requirements.

Neural Network Model Parser: The entry point of the fpgaHART toolflow is the Model Parser. To facilitate the process of incorporating various neural network (NN) models into the toolflow, a dedicated NN model parser has been developed. This parser is designed to read and map NN models in the ONNX format, a standardised format supported by many deep learning frameworks such as PyTorch and TensorFlow, into an intermediate representation (IR) format required by the toolflow. This IR, can be described as a Directed Acyclic Graph (DAG), as detailed in Section 3.3.1, serving as an abstracted form of the network architecture and capturing the key details about the model's layers, connections, and dataflow patterns, enabling subsequent stages to optimize the design for FPGA execution.

Design Space Exploration (DSE) Optimiser: After generating the intermediate representation (IR), it is translated into a Synchronous Data-Flow Graph (SDFG), denoted as G, as outlined in Section 3.3.2. The core concept of synchronous dataflow modelling is that each node fires whenever data is available at its inputs, resulting in a paradigm of data-driven execution. This format is compatible with the rest of the toolflow's tools, such as the optimiser and the toolflow's resource and performance models. The graph G is then passed to the Automated Design Space Exploration (DSE) component, which serves as the optimization engine. The DSE optimiser explores various configurations and partitioning strategies, such as selecting optimal reconfiguration points, managing parallelism within partitions, and balancing computational workloads. It also considers the FPGA platform specifications and application performance requirements, allowing for an informed trade-off between performance metrics such as throughput and resource utilization. The tool is able to generate hardware designs that are highly optimized and tailored to the specific model characteristics.

Network Hardware Mapping: The final stage of the fpgaHART toolflow is the Network Hardware Mapping, where the optimized IR, expressed as G, is converted into hardware-specific implementations. This involves generating the Hardware Description Language (HDL) code based on the optimized design configurations, and the respective RTL code which is then used for the generation of the final bitstream.

The fpgaHART toolflow aims to streamline the process of deploying 3D CNN models on FPGAs by automating key steps in model translation, optimization, and hardware mapping. Its design supports high-performance, throughput-oriented applications by leveraging FPGA reconfiguration capabilities, making it suitable for a range of FPGA platforms and various application domains. The flexibility to adapt partitioning strategies and optimize hardware mappings based on specific performance and resource constraints enables significant improvements in efficiency and adaptability compared to traditional approaches.

4.3.1 Extra DAG Nodes and Hardware Building Blocks

To support the diverse architectures of the models introduced in this chapter (C3D, R(2+1)D, and SlowOnly), we must extend the set of supported Directed Acyclic Graph (DAG) nodes and their corresponding hardware building blocks, as they have been defined in Section 3.3.1 for the X3D model. While Section 3.3.1 establishes the core primitives required for X3D, the models evaluated here utilize supplementary layer variants that are essential for their successful mapping and optimisation within our toolflow.

3D Convolution Layers

These new modules include several variants of 3D Convolution (Full Convolution (3D), Spatial and Temporal 3D Convolution). While the general concept of 3D Convolution was introduced in Section 2.1.2, here we define some variants not previously defined:

- **Spatial Convolution (2D-like)** is a special case of 3D convolution where the temporal dimension of the kernel is fixed to 1, resulting in a kernel shape of $K_h \times K_w \times 1$. This simplifies the operation to a 2D convolution applied across the spatial dimensions (height and width) only, while leaving the temporal dimension unchanged.

- **Temporal Convolution (1D-like)** is another special case of 3D convolution where the spatial dimensions of the kernel are fixed to 1, resulting in a kernel shape of $1 \times 1 \times K_d$. This reduces the operation to a 1D convolution applied only along the temporal dimension, with no changes to the spatial dimensions.

We have extended the set of supported Directed Acyclic Graph (DAG) nodes and their corresponding hardware building blocks by incorporating the variants of the 3D Convolution operation as described above. While the 3D Convolution node retains its high-level representation within the DAG nodes and HW building blocks, as described

in Sections 3.3.1 and 3.3.2, its underlying hardware implementation has been refined to optimize performance for specific cases present in these models.

3D Pooling Layers

There are two different types of pooling layers, as described in Section 2.1.1: (a) 3D max pooling, (b) 3D average pooling. Even though the are different types of pooling, the discrimination between them can be achieved by looking at the layer's configuration that is stored in the DAG nodes, $< \mathbf{S_n^{in}}, \mathbf{S_n^{out}}, \mathbf{J_n}, \mathbf{P_n}, \mathbf{Gr_n} >$:

- $\boldsymbol{J}_n$ is a 3-value vector $[J_n^h, J_n^w, J_n^d]$ specifying the strides of the pooling along each dimension.
- $\boldsymbol{P}_n$ is a 6-value vector $[P_n^{h_s}, P_n^{h_e}, P_n^{w_s}, P_n^{w_e}, P_n^{d_s}, P_n^{d_e}]$ denoting the amount of padding applied at the start and end of each dimension of the input.
- $\boldsymbol{T}_n$ denotes the type of pooling operation, e.g. max or average.

Similarly, we introduced a dedicated hardware building block for 3D Pooling layers. The parameters of the 3D Pooling hardware building block are consistent with descriptions in Chapter 3.5.3. Each pooling layer's hardware node includes both DAG parameters, and SDFG parameters.

3D Pooling - $< \boldsymbol{DAG_n^{params}}, \boldsymbol{c_n^{in}}, \boldsymbol{c_n^{out}}, \boldsymbol{r_n^{in}}, \boldsymbol{r_n^{out}} >$ During the Design Space Exploration (DSE), the parameters $\boldsymbol{c_n^{in}}$, $\boldsymbol{c_n^{out}}$ are adjusted, impacting the final performance of each layer. Meanwhile the $\boldsymbol{r_n^{in}}$ and $\boldsymbol{r_n^{out}}$ depend on the $\boldsymbol{c_n^{in}}$ and $\boldsymbol{c_n^{out}}$ which means they are implicitly altered during the DSE as well.

4.3.2 Partitioning Strategies

Partitioning plays a critical role in optimizing FPGA-based acceleration for throughput-oriented designs, leveraging the reconfiguration capabilities of FPGAs to achieve higher performance. As previously discussed in Section 3.4.1, streaming architectures may be limited by FPGA resources due to their nature of optimising each specific layer of the model independently. In the previous chapter, the partitioning for the X3D model was performed manually, with reconfiguration points selected based on an analysis of the model's layers and blocks. In contrast, the fpgaHART toolflow automates this process, refining the resolution of partitioning and further expanding the design space. The tool allows the user to provide initial hints and heuristics to guide the optimization and partitioning process. Specifically, the user can provide a starting estimate for the number of reconfiguration points, L, which serves as a seed for the algorithm to begin exploring.

This is not a strict constraint but rather a flexible guideline that helps define the initial scale of the number of partitions. Additionally, the user may specify the types of layers where partitioning is permitted—for instance, allowing splits only after convolutional layers or before element-wise layers. This enables the incorporation of the user's domain knowledge into the optimization process, leveraging their hints to make the optimization process more targeted and efficient. By providing finer control and automating the search for optimal reconfiguration points throughout the entire model architecture, fpgaHART enhances the granularity of partitioning compared to the coarse-grained, manual approach employed in the previous chapter. The number of partitions and their placement are now adjustable parameters that the optimization algorithm can refine to achieve optimal performance.

However, allowing the tool to split the computation graph at any arbitrary point, if no hints are given, introduces several challenges.

Challenge 1: The expanded design space becomes considerably larger, as the reconfiguration points are now introduced as an additional parameter in the optimization process. This increase in potential configurations allows for greater flexibility in achieving optimal performance but also adds complexity to the Design Space Exploration (DSE) process, requiring more sophisticated algorithms or heuristics to efficiently navigate the expanded search space within feasible time and resource constraints.

To address this challenge, the proposed optimization process incorporates user-provided hints to guide the exploration of the design space, starting with an initial partitioning of the model's layers by selecting L initial reconfiguration points. These points are determined based on the user-specified allowed layer types for partitioning the model, when such hints are provided; otherwise, default values are utilized. The partitions are obtained by randomly selecting L allowed layers and splitting the graph into $L + 1$ initial partitions. These initial partitions provide an initial state of the model that the optimiser refines. The refinement process focuses on two key principles:

- **Identifying Invalid Partitions**: Invalid partitions are those that surpass the available FPGA resources, even when the layers are configured with the minimum level of parallelism. This typically occurs when partitions contain branches or multiple branches that require extensive buffering, often surpassing the BRAM capacity of the FPGA, or when partitions contain layers with parameters that exceed the on-chip storage capacity of the device. Most instances of invalidity arise from excessive BRAM utilization, as the validity check is performed with the minimum level of parallelization, which utilizes only a small number of DSPs

When an invalid partition is identified, the algorithm resolves the issue by further splitting the partition, randomly selecting an allowed layer within the partition to create additional sub-partitions. In extreme cases, where a partition consists of a single layer that cannot fit on the FPGA, the optimizer is unable to further split the partition, and the process terminates. In such situations, the use of a larger FPGA device is required. This whole process ensures that the refinement can continue efficiently, mitigating resource constraints wherever possible.

- **Identifying Redundant Partitions**: Beyond addressing excessively large partitions, the optimization process also considers very small partitions that can emerge during the initial partitioning of the model. These small partitions typically consist of only a few layers with minimal parameters to store and no branching, resulting in resource requirements that are so low they can be merged with adjacent partitions without significant impact. When such partitions are identified, they are marked as candidates for merging. The algorithm then evaluates their adjacent partitions to determine if merging is feasible. This decision is guided by heuristics, ensuring that the resulting merged partition does not exceed some predefined resource thresholds. These thresholds are hyperparameters of the optimization algorithm, allowing users to explicitly set and fine-tune them. By merging redundant partitions, the process minimizes the reconfiguration overhead associated with maintaining unnecessary partitions, leading to a more efficient overall design.

The algorithm of the automated partitioning is provided in Algorithm 2.

Once the partitions have been validated and any redundant partitions have been merged, they are considered static and fixed for the remainder of the optimization process. At this stage, the the SA optimiser, as described in Algorithm 1, applies a series of partition-specific optimization steps, as described in Section 3.4.3, employing both coarse and fine transformations to the SDFG and exploring the maximum level of parallelism achievable within the constraints of the available FPGA resources. Once the optimizer has converged for all the partitions, the final performance of the model is evaluated based on Equation (3.6).

Challenge 2: Arbitrary partitioning can lead to complex partition structures. Modern 3D CNN architectures often include complex structures with numerous skip connections and branching layers, which complicates the partitioning process. For example, branches may be split before the merging point, resulting in partitions with multiple input and output channels. These multi-input and multi-output partitions

Algorithm 2 Automated Partitioning Algorithm

Input: G, R_{FPGA}, L, $L_{allowed}$
Output: P_{final}
$P \leftarrow$ InitialPartitions$(G, L, L_{allowed})$ ▷ Randomly create $L + 1$ initial partitions
changed $\leftarrow$ true
while *changed* **do**
    changed $\leftarrow$ false
    for each partition p in P **do**
        if IsInvalid(p, R_{FPGA}) **then** ▷ Check partition resources with min. parallelism
            if $|p| > 1$ **then**
                $P \leftarrow$ SplitPartition$(P, p, L_{allowed})$ ▷ Further split the invalid partition
                changed $\leftarrow$ true
                break ▷ Restart loop as partition set P has changed
            else
                Terminate ▷ Sinlge-layer partition exceeds resources
    if *changed* **then continue**
    for each partition p in P **do**
        if IsRedundant(p) **then** ▷ Check for very small partitions
            if CanMergeWithAdjacent(p, R_{FPGA}) **then** ▷ Check feasibility based heuristics
                $P \leftarrow$ MergePartition(P, p) ▷ Merge with an adjacent partition
                changed $\leftarrow$ true
                break ▷ Restart loop as partition set P has changed
$P_{final} \leftarrow P$
return P_{final}

create additional demands on memory bandwidth, requiring careful coordination to manage off-chip memory access and avoid bottlenecks. Furthermore, balancing the workload across these fragmented partitions is essential to maintain overall throughput, as imbalanced partitions can lead to inefficiencies and underutilized FPGA resources.

To address the second challenge, we designed a set of splitting strategies to ensure that partitions are structured to minimize memory overhead and optimize dataflow, while also enabling a more manageable partitioning outcome. Below we present the three partitioning strategies that we employ during graph splitting:

- **Split After Branch Merging Point**. This approach involves splitting the model graph immediately after a branch merging point, thereby keeping each branch intact. To ensure the branches are correctly handled and synchronized at the merging layer, a FIFO buffer is introduced to the branch with the higher processing rate. The size of this buffer is determined by analysing the production and consumption

rates of each layer in both branches. Utilising this information, along with the workload each layer processes and the level of parallelism employed, the rates at the merging point for both branches can be inferred. Based on these rates, the buffering requirements for the branch with the higher rate are calculated to prevent potential deadlocks or stalls in the pipeline. By preserving the full structure of the branches, this strategy facilitates parallelism while maintaining efficient dataflow across partitions. A visualization of this splitting strategy is depicted in Figure 4.3.

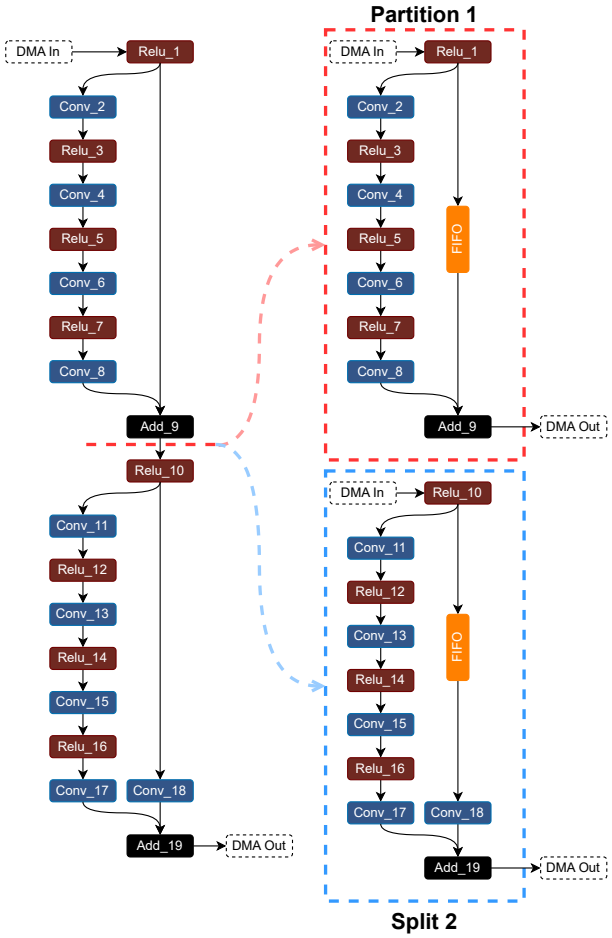


Fig. 4.3 Partitioning the model graph after a branch merging point. The branches remain intact, with a FIFO buffer added to the higher-rate branch to synchronize dataflow and prevent pipeline deadlocks.

- **Split Before Branch Merging Point or Mid Branch**. This partitioning strategy involves dividing the model graph either before or within a branch. By splitting before the branch merging point or mid branch, the partitioning generates at least one sequential-like segment, which reduces on-chip memory usage as no additional buffering is required to synchronize the branches. The first partition will have multiple outputs aligned to the branches, and the second will have multiple inputs to feed the merging node. This approach introduces a multi-input and multi-output partitions, requiring multiple off-chip memory connections. Special care must be taken to manage memory bandwidth effectively to prevent bottlenecks and ensure efficient dataflow across partitions. A visualization of this splitting strategy is depicted in Figure 4.4.

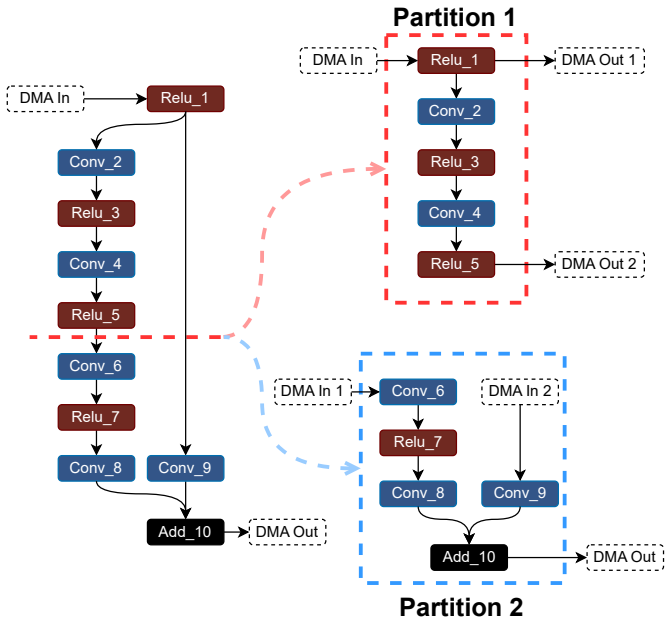


Fig. 4.4 Partitioning before the branch merging point or mid branch generates two partitions: one multi-output sequential partition and one multi-input partition that includes the merging node. This approach reduces on-chip memory usage by avoiding extra buffering, while the multi-input, multi-output partitions require careful management of memory bandwidth due to multiple connections to off-chip memory.

- **Split at "Bottleneck" Layers**. This strategy targets layers within the model that are known to cause performance degradation, such as bottleneck layers. These layers are specified by the user through hints on allowed layers for graph partitioning, as outlined in Challenge 1 above. For instance, the global average pooling layer

in the squeeze-and-excitation module of the X3D model can significantly slow down processing. By splitting the graph at these bottleneck layers, the output of the problematic layers is directed straight to off-chip memory, allowing the next partition to load the data after reconfiguring the FPGA. This approach enables the bypassing of bottlenecks but also requires careful management of memory bandwidth, as both multi-input and multi-output partitions may be generated. A visualization of this splitting strategy is depicted in Figure 4.5.

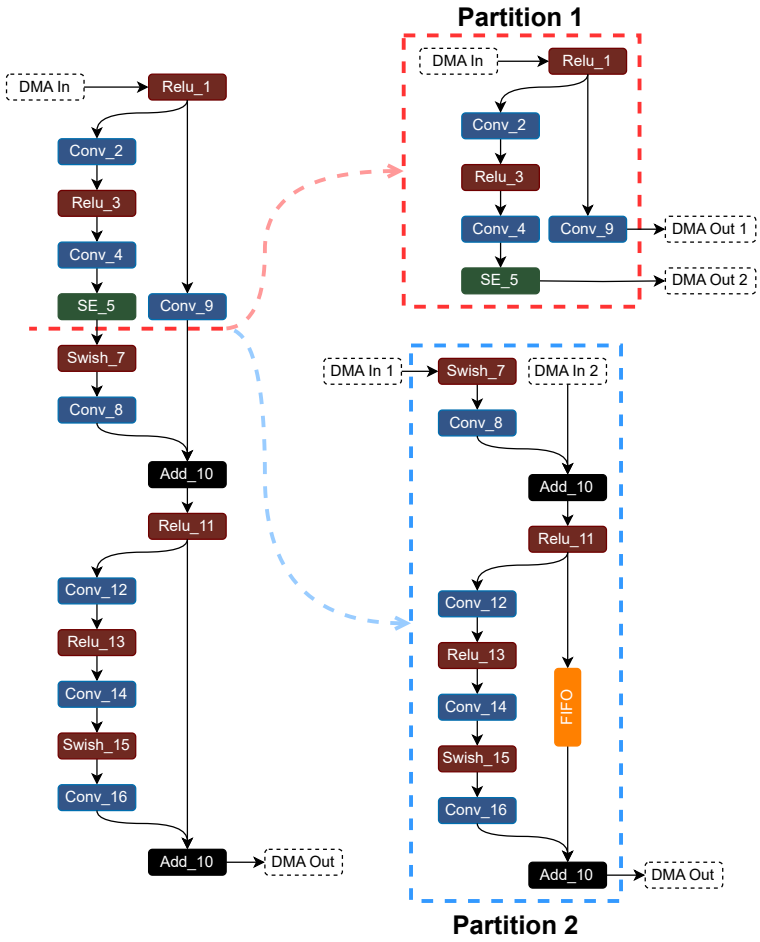


Fig. 4.5 Partitioning at bottleneck layers directs output data straight to off-chip memory for the next partition. This strategy helps bypass performance bottlenecks but requires managing the memory demands of multi-input and multi-output partitions.

These partitioning strategies allow the optimiser to efficiently handle complex architectures, ensuring that skip connections and branching patterns are maintained or organized in a way that enhances throughput and resource utilization.

Figure 4.6 illustrates the overall execution timeline for a model that has been partitioned into 3 partitions. The process begins with the initial configuration of the FPGA with the bitstream for the first partition, $\mathcal{P}_1$. Once configured, the entire batch of B input samples is processed by this partition. After the first partition completes its workload, the FPGA is fully reconfigured with the bitstream for the second partition, $\mathcal{P}_2$, which incurs a reconfiguration overhead. This cycle of processing and reconfiguration continues for all subsequent partitions until the final partition, $\mathcal{P}_3$, completes its execution. The total execution time for the entire batch is the sum of the execution latencies for all partitions and the cumulative reconfiguration overhead, as previously defined in Equation (3.6). This timeline visually demonstrates how the reconfiguration penalty is amortized over a large batch size, making this approach highly effective for throughput-oriented applications.

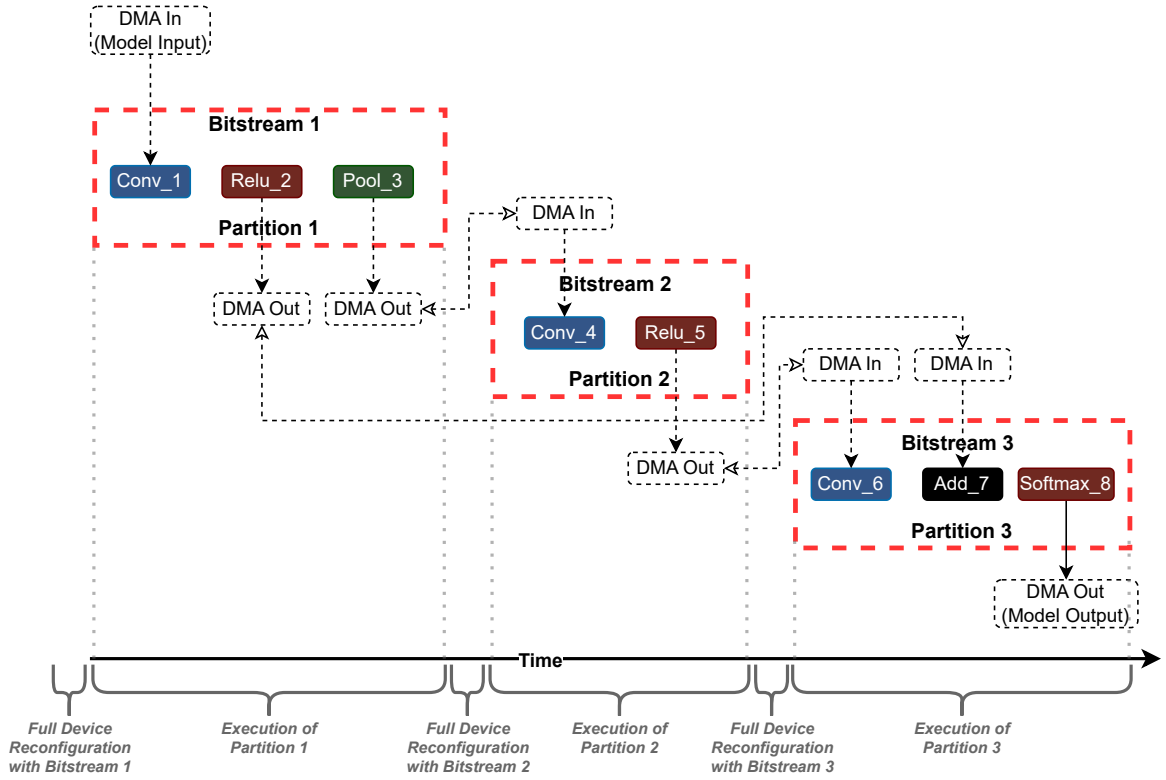


Fig. 4.6 Overall execution timeline for a model divided into P partitions. The total latency is the sum of each partition's execution latency and the reconfiguration time between partitions.

4.4 Performance Evaluation and Results

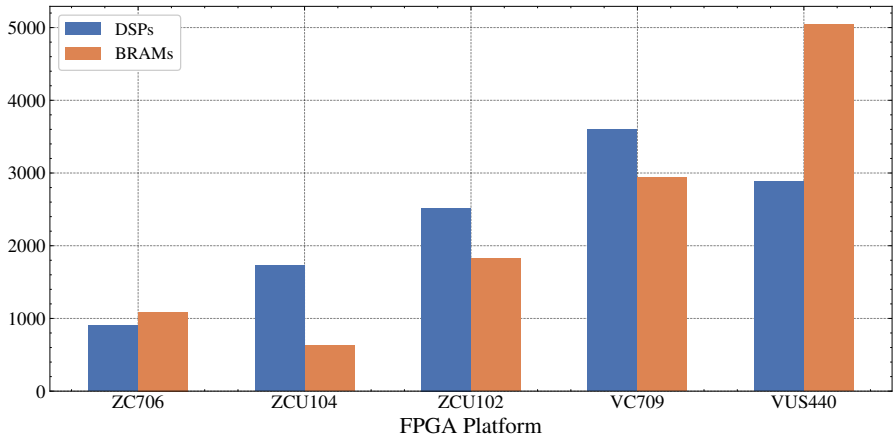
4.4.1 Experimental Setup

To evaluate the performance of the tool, four state of the art 3D CNN HAR models have been selected, Slowonly, R(2+1)D-18, R(2+1)D-34, and X3D (as shown in Table 4.2), five AMD/Xilinx boards: the ZC706 (Zynq-7000 XC7Z045), ZCU104 (Zynq UltraScale+ XCZU7EV), ZCU102 (Zynq UltraScale+ XCZU9EG), VC709 (Virtex-7 XC7VX690T), and VUS440 (Virtex UltraScale XCVU440), to demonstrate the ability of the tool to target multiple 3D CNN with different workloads and network parameters on a variety of platforms. The justification behind choosing the specific 3D CNN models is already discussed in Section 4.2. The specific devices were chosen as evaluation platforms to represent a hierarchical scaling of resources, ranging from smaller-scale devices like the XC7Z045 to larger-scale devices such as the XCVU440, as illustrated in Figure 4.7. This selection allows us to comprehensively evaluate the performance and scalability of our proposed designs across target platforms of varying resource capacities, as shown in Figure 4.8. The C3D model was also included to provide direct comparisons with existing works, the majority of which are hand-tuned, model-specific architectures and not toolflows. Vitis HLS and Vivado Design Suite (2021.2) were used, while the reported resource results are after place and route at 160 MHz clock frequency. The arithmetic precision used was 16-bit fixed point arithmetic with $Q8.8$ format. The accuracy of the HAR models is evaluated on UCF-101, following the same strategy as prior studies [1, 97]. The ONNX models used for all the experimental results have been exported from [122] for *C3D, Slowonly, and X3D-M* models, and from [123] for *R(2+1)D-18 and R(2+1)D-34*.

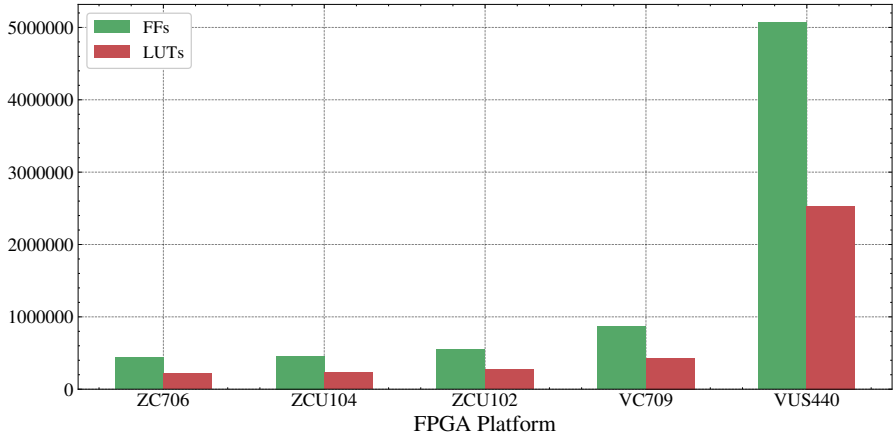
4.4.2 Performance Evaluation

Accuracy of Modelling

To evaluate the quality of the performance predictor a series of experiments was conducted. Four partitions were chosen to cover the variety of produced graph structures (i.e. branch, sequential, multi-inputs, multi-outputs). The relative error was used to measure the difference between the predicted and actual throughput. The relative errors for the four aforementioned types are 12.89%, 5.03%, 11.92%, and 17.32% respectively, giving



(a) DSPs and BRAMs per FPGA platform.



(b) FFs and LUTs per FPGA platform.

Fig. 4.7 Resource availability across different FPGA platforms: (a) Comparison of DSPs and BRAMs, showcasing the computational and memory capabilities; (b) Comparison of FFs and LUTs, highlighting the logic resources available.

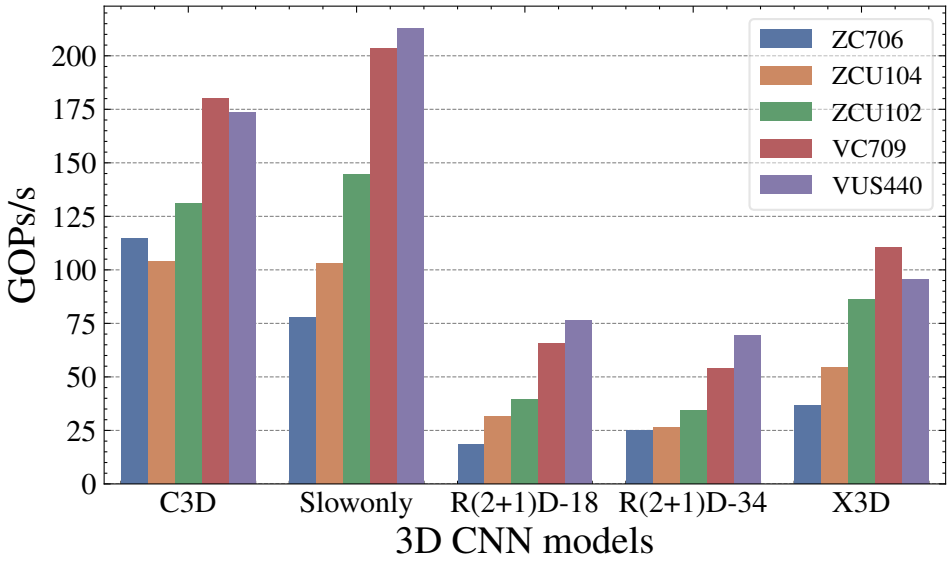


Fig. 4.8 Throughput (GOPs/s) of fpgaHART-generated designs on 3D CNN HAR models delivering high-throughput results on a variety of FPGA devices

a geometric mean relative error of 10.75%[3]. We found that the above errors are small enough to lead to meaningful design space exploration.

Performance Evaluation

Figure 4.8 displays the performance in GOPs/s (with a batch size of 100) of the fpgaHART-generated designs for the 3D CNN models of Table 4.2, which details their unique characteristics, on a variety of FPGA devices. Such batch sizes are frequently encountered in practise when generating multiple views and clips over time and averaging them to improve the performance of the predictions, as previously described in Section 2.3.3. Even larger batch sizes may be required for multi-person HAR systems that evaluate each person's actions independently, as well as for large-scale systems that simultaneously analyse several videos. As expected, the performance of the proposed designs correlates with the available resources of the targeted FPGA devices, following a similar trend to the hierarchical scaling of resources across the platforms (Figure 4.7).

To provide a more detailed runtime profile, Table 4.3 presents a comprehensive breakdown for the C3D model executing on the ZCU102 platform. The model is divided into seven distinct partitions by our automated strategy, each with a different number of layers and computational workload. The table details the performance of each

[3]geometric mean was used as these structures can be found simultaneously in a single 3D CNN

partition, showing variance in metrics like latency and throughput, which reflects how the optimiser balances the workload against resource constraints. Crucially, the table quantifies the impact of batching on performance. For a batch size of 1, the total latency is 0.7878 seconds. This includes the compute time and the reconfiguration overhead between partitions. As the batch size increases to 100, the overall throughput improves significantly from 49.13 GOPs/s to 130.84 GOPs/s. The values in the "Latency (s)" and "Throughput (GOPs/s)" columns are calculated based on Equations (3.4) to (3.6). This demonstrates how the fixed cost of reconfiguration time is effectively amortised across a larger number of inferences, validating the use of large batch sizes in our throughput-oriented evaluations.

Table 4.3 Detailed runtime profile of the fpgaHART-generated C3D accelerator on the ZCU102 platform. The table provides a partition-level breakdown of key performance metrics and illustrates the effect of different batch sizes on total latency and throughput, highlighting the amortisation of reconfiguration overhead.

Partition	Num Layers	Cycles	Pipeline Depth (Cycles)	GOPs	Latency (s)	Throughput (GOPs/s)
1	7	12845056	159353	12.158	0.0812	149.58
2	2	3612672	41488	5.549	0.0228	242.96
3	3	5724310	22678	11.1	0.0359	309.03
4	2	3150384	14864	2.775	0.0197	140.27
5	3	6855044	8540	5.549	0.0428	129.35
6	2	7173166	3490	0.791	0.0448	17.63
7	8	7187593	5551	0.791	0.0449	17.61
Total Sum (Avg)	27 (3.8)	46548225 (6649746.4)	255964 (36566.2)	38.61 (5.5)	0.2925 (0.0417)	1006.45 (143.77)
Batch 1	27	46548225	255964	38.61	0.7878	49.13
Batch 30	27	1396446750	255964	38.61	9.2246	125.91
Batch 100	27	4654822500	255964	38.61	29.5895	130.84

The placement of fpgaHART in comparison to the rest of the existing works is outlined in Table 4.4, where the fpgaHART results are reported using ZCU102 as the FPGA board. A conclusion readily apparent from Table 4.4 is that fpgaHART is capable of delivering competitive performance on several 3D CNNs that have not been previously addressed and have a broad set of workloads and network parameters.

Figure 4.9 presents the current state of the Pareto front expressed in terms of Top-1 accuracy (%) over throughput (clips/s), where the fpgaHART generated designs were derived targeting the VC709 FPGA board. The results demonstrate that the fpgaHART models have pushed the Pareto front, delivering solutions that balance high throughput

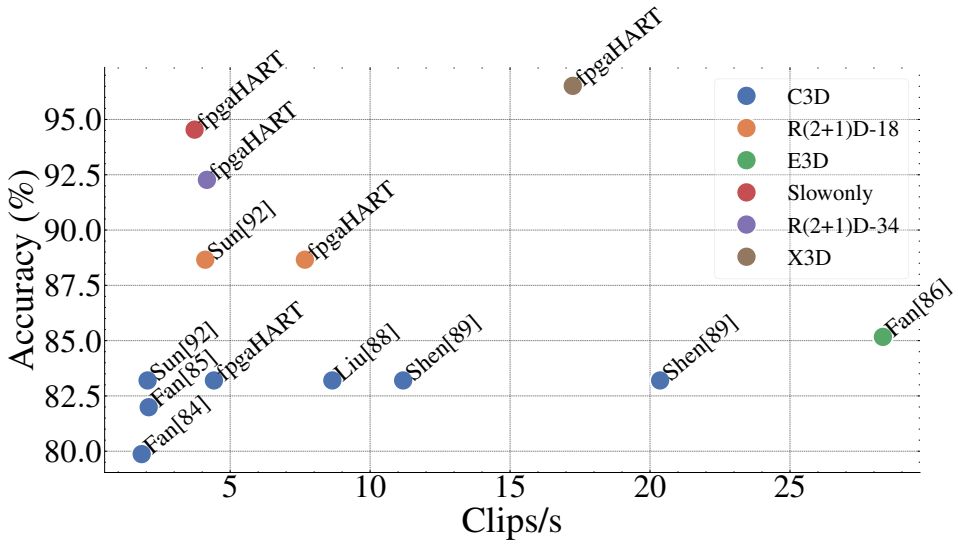


Fig. 4.9 Pareto front on 3D CNNs: Throughput (Clips/s) over Top-1 Accuracy (%). The fpgaHART results were taken using the VC709 FPGA board, delivering solutions on the Pareto front.

and high accuracy, as illustrated in the graph. A detailed discussion is provided below, organized by model type:

- **C3D:** The designs of fpgaHART demonstrate competitive throughput compared to the rest of the existing works for the C3D model. Specifically, the proposed design achieves 2.39×, 2.11×, and 2.15× higher throughput than Fan et al. [109], Fan et al. [110], and Sun et al. [116], respectively. However, compared to Liu et al. [113], our design falls short by a factor of 1.96. A comparison with Shen et al. [2] is not feasible as they utilize a different, smaller variant of the original C3D model [3] used in the other works.

- **R(2+1)D-18 and R(2+1)D-34:** Compared to the only prior work targeting this model by Sun et al. [116], our proposed solution achieves 1.86× higher throughput. For the R(2+1)D-34, a larger variant of the same architecture, the throughput, as expected, decreases by 1.83× times compared to its smaller counterpart, but this is accompanied by a notable improvement in Top-1 Accuracy, which increases by 3.61*pp*.

- **E3D:** This model, introduced by Fan et al. [111], was developed through a co-design approach that tightly couples the architecture of the 3D CNN model with the characteristics of the underlying FPGA design. As a result, the hardware and model

are highly optimized for each other. Unfortunately, the authors have not made the model's weights or architecture publicly available, making a direct comparison impossible.

- **SlowOnly:** A direct comparison for this model is not available. The SlowOnly model is a high-performance architecture, achieving a Top-1 Accuracy of 94.54%. However, its heavy computational workload of 54.9 GMACs limits its throughput to 3.72 clips per second.
- **X3D:** The X3D model is another high-performance architecture, achieving a Top-1 Accuracy of 96.52%. Its efficient design and relatively small computational workload enable a throughput of 17.24 clips per second. Apart from the designs discussed in the previous chapter, there are no other prior works targeting this network. A detailed comparison with the designs of Chapter 3 is provided later in this section.

The results collectively showcase fpgaHART's ability to push the Pareto front across different model types, delivering solutions that achieve both high accuracy and high throughput tailored to the unique characteristics of each model.

Comparison versus GPU

Comparing the results for the C3D model (batch size 30, targeting the ZCU102 board) to the Nvidia RTX 2060—a mid-range GPU with 1920 CUDA cores and a 1.2 GHz clock speed—the proposed architecture achieves a throughput of 4.42 clips/s, whereas the GPU delivers 43.49 clips/s. However, the proposed solution consumes only 26 W, significantly less than the GPU's 77.28 W (excluding the additional CPU power required by a GPU system), resulting in an efficiency of 0.17 clips/s/watt compared to the GPU's 0.56 clips/s/watt. The GPU power estimates were obtained using the *nvidia-smi* tool, while the FPGA power estimates were derived from Vivado tool estimations.

X3D Comparison Across Chapters

When comparing the throughput performance of the X3D model between this chapter and the previous one, a modest decline can be observed with throughput dropping from 18.72 clips/s in Chapter 3 to 13.44 clips/s in the current chapter. This difference is due to various methodological adjustments made in this chapter's automated optimisation process, which, while resulting in slightly lower throughput in this situation, does not undermine the effectiveness of the fpgaHART toolflow for a variety of reasons.

To begin, the automated Design Space Exploration (DSE) used in fpgaHART presents a broader and more flexible search method, systematically exploring reconfiguration points throughout a much larger design space. Unlike the manual partitioning employed in the previous chapter, this automated approach optimises across multiple configurations using a heuristic algorithm, which can occasionally result in a local minima. As a result, even though the high-performance configuration identified manually in Chapter 3 is part of the design space, it may have not been examined during the DSE, leading to the noted throughput difference. However, this automated approach is advantageous for scaling across various architectures, even if it introduces slight variations in results.

Furthermore, the refined performance and resource models used in this chapter have some effect on the DSE outcomes. These models are more accurate in predicting performance and resources, but they differ from those employed in Chapter 3, making a direct comparison across chapters quite challenging.

Secondly, the refined performance and resource models introduced in this chapter play a significant role in shaping the DSE outcomes. These models offer improved accuracy in predicting performance metrics, but they differ from those used in Chapter 3, making a direct comparison between chapters invalid. Even if in this specific case of X3D the updated models yield slightly lower throughput predictions, they provide more precise insights into the design space.

In conclusion, while the throughput for X3D in this chapter is slightly lower, this discrepancy reflects the current approach's greater adaptability and precision, across diverse models and FPGA platforms. The advances gained in automating the optimisation process and refining prediction models contribute to fpgaHART's long-term potential for high-throughput applications, even if minor variations in performance may arise in certain cases, such as the X3D model.

Table 4.4 Comparison with existing works on 3D CNN HAR models

	H. Fan [109]	H. Fan [110]	Z. Liu [113]	J. Shen [2]†		M. Sun [116]		H. Fan [111]	fpgaHART (Ours)				
Model	C3D	C3D	C3D	C3D		C3D	R(2+)D-18	E3D	C3D	Slowonly	R(2+1)D-18	R(2+1)D-34	X3D
GMACs	38.61	38.61	38.61	-		38.61	8.52	6.1	38.61	54.9	8.52	12.91	6.97
Precision	W16A16	BFP24	W16A16	W16A16	W16A16	W16A16	W16A16	Float32	W16A16	W16A16	W16A16	W16A16	W16A16
Accuracy (%)	79.87	81.99	83.2	83.2		83.2	88.66	85.17	83.2	94.54	88.66	92.27	96.52
Board	ZC706	ZC706	VC709	VC709	VUS440	ZCU102	ZCU102	Intel SX660	ZCU102	ZCU102	ZCU102	ZCU102	ZCU102
Reconfigurations∓	-	-	-	-	-	-	-	-	6	30	20	43	37
DSP (%)	90	86.6	99.8	42	53	48	48	93.3	60.95 (51.49)	91.74 (63.77)	92.81 (66.21)	94.81 (66.46)	94.96 (84.43)
BRAM-18K (%)	86.6	88.1	26.6	52	30	100	100	-	94.73 (91.51)	94.73 (78.22)	94.73 (78.09)	94.73 (84.07)	94.73 (52.71)
Frequency (MHz)	172	200	120	150	200	150	150	150	160	160	160	160	160
Throughput (clips/s)*	1.84	2.09	8.65	11.18	20.36	2.05	4.11	28.32	3.38	2.54	4.62	2.63	13.44
GOps/s*	70.41	80.12	330.74	427.29	778	78.44	111.71	172.8	130.84	144.44	39.59	34.26	85.96
GOps/s/DSP*	0.087	0.103	0.092	0.281	0.511	0.065	0.092	0.109	0.052	0.057	0.015	0.013	0.034
Op/DSP/cycle*	0.511	0.519	0.774	1.874	2.559	0.435	0.613	0.727	0.325	0.358	0.098	0.084	0.213

* Batch size 100. † The C3D model used is different/smaller version from the original one [3].

∓ Designs with '-' indicate no extra reconfiguration of the device after the initial bitstream in loaded

4.5 Conclusions

In this chapter, we introduced fpgaHART, a throughput-oriented toolflow designed to automate and optimize the deployment of 3D Convolutional Neural Networks (CNNs) on Field-Programmable Gate Array (FPGA) platforms. Targeted for high-throughput applications, fpgaHART expands the design space by enabling the exploration of optimal reconfiguration points and supporting finer-grained partitioning options during model partitioning, while also automating the process to streamline optimization and deployment. The main contributions of this chapter include:

- Development of fpgaHART Toolflow: fpgaHART was introduced as a comprehensive solution for mapping 3D CNNs onto FPGAs, capable of adapting to multiple models and FPGA platforms. This framework extends beyond model-specific optimization, offering a versatile tool for various 3D CNN architectures targeting throughput-oriented applications.

- Automated Model Partitioning and Optimisation: The toolflow incorporates an automated partitioning mechanism that identifies optimal reconfiguration points, allowing for per-partition optimizations. This approach not only significantly reduces the need for manual partitioning, which traditionally posed scalability challenges, but also facilitates the exploration of the expanded design space.

- Performance Evaluation Across Models and Devices: We rigorously evaluated fpgaHART on multiple FPGA platforms and a variety of 3D CNN models, demonstrating its adaptability and high throughput potential. The toolflow proved effective in achieving competitive performance across diverse architectures, establishing a new standard for scalability in FPGA-based 3D CNN deployment.

With fpgaHART, we developed a throughput-oriented solution that not only reduces the time and resource costs of manual optimization but also adapts to complex 3D CNN structures with branching and skip connections. While fpgaHART demonstrates substantial advancements in throughput and resource efficiency, certain challenges remain. The expanded design space introduced by automated partitioning requires sophisticated exploration strategies, which can impose higher computational resource and time demands. Moreover, the emergence of multi-input, multi-output partitions elevates the importance of carefully managing off-chip memory bandwidth (BW) and on-chip storage to maximize

model performance. Fine-tuning these constraints holds significant potential for further gains in throughput and efficiency, marking an exciting direction for future research.

The next chapter investigates these challenges with a focus on optimizing the trade-off between off-chip memory Bandwidth (BW) and on-chip storage, a crucial factor in balancing bandwidth and storage demands. This enhancement aims to push the performance-accuracy frontier for FPGA-accelerated CNN applications.

Chapter 5

Expanding Horizons: Enhancing CNNs Performance by Leveraging Off-Chip Memory Utilisation on FPGAs

5.1 Introduction

Accelerating CNNs on FPGAs usually utilises one of the two main approaches: (1) designing a unified accelerator architecture and (2) implementing a streaming dataflow architecture. The unified accelerator approach acts as a "one-size-fits-all" system, aiming to minimise engineering effort by accommodating multiple types of networks within a single design [124, 58]. However, this technique frequently falls short of fully exploiting on the unique characteristics of each model and the resource availability of FPGA devices, limiting its efficiency primarily because this architecture style is not tailored to a specific model. In contrast, the streaming dataflow design tailors a layerwise pipeline to each unique network workload, maximising performance under a specific network configuration. This approach has proven particularly effective for applications where the model remains unchanged over time.

Within the streaming architecture paradigm, current research has mostly focused on optimising computation engines and on-chip memory. Tools like fpgaConvNet [24], FINN [23], HLS4ML [26] and HPIPE [25] have contributed to automating the generation of High-Level Synthesis (HLS) or Register Transfer Level (RTL) implementations from

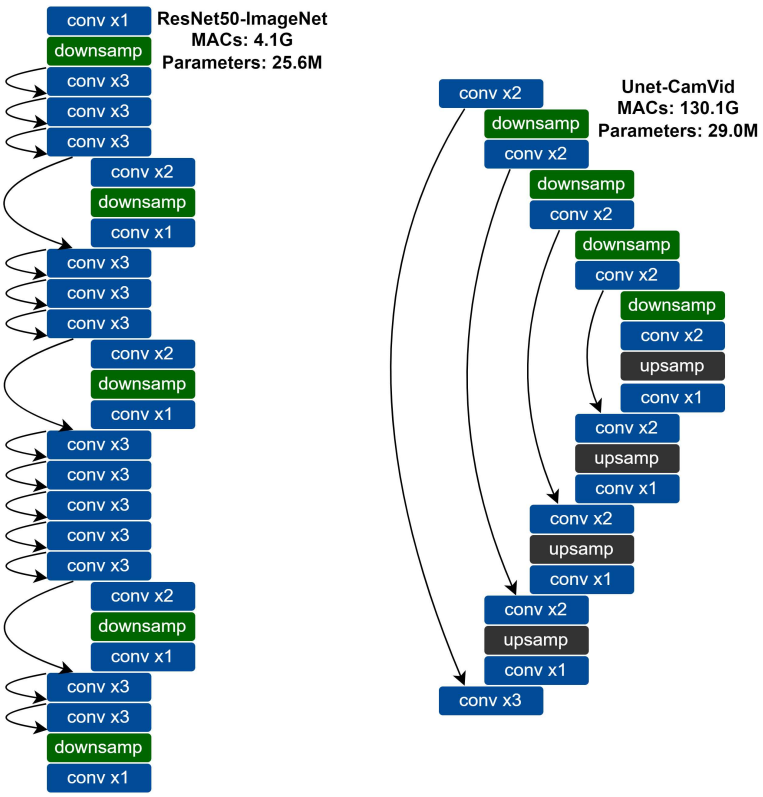


Fig. 5.1 Model structures of ResNet50 on the ImageNet classification task [4] and UNet on the CamVid segmentation task [5]. These two models are similar in the number of weight parameters. However, UNet requires much more computation effort due to a high input resolution and its lengthy and hierarchical branch connections pose a unique challenge for streaming architectures.

high-level network descriptions such as ONNX or PyTorch. A typical approach in these frameworks confines data movement within the FPGA chip: weights are kept statically in on-chip read-only memory, and activations are carried between layers via First-In-First-Out (FIFO) buffers and handshake interfaces. This strategy, while effective for small networks, has significant scalability limitations as models become more sophisticated and complex.

For example, the widely adopted UNet [36] architecture for semantic segmentation clearly demonstrates these challenges as show in Figure 5.1. With 130.1 GMACs and a highly branched, hierarchical structure, UNet necessitates significant buffering and synchronisation of data streams, which frequently exceeds available on-chip memory. Similar challenges arise in more modern networks, such as YOLO [125] and X3D [6], which have millions of parameters and very deep skip connections, spanning up to 60

layers. These characteristics impose significant constraints when mapping such models onto FPGA platforms.

Existing streaming architectures primarily focus on optimizing computation engines and on-chip memory resources but they are often constrained by the limited capacity of on-chip memory. The applicability of existing toolflows becomes limited when on-chip memory demands exceed the available resources. However, in such architectures, off-chip memory is primarily used for data transfers at the input and output pipeline stages, meaning its utilization remains relatively low. This presents an opportunity to better leverage off-chip memory when on-chip memory becomes a limiting constraint, hindering the applicability of such architectures.

This chapter presents a novel memory optimisation strategy rethinking the way both on-chip and off-chip memory are utilised in layerwise pipelined, streaming architectures. Building upon the toolflow presented in Chapter 4 (Figure 4.2), it expands it to target a broader design space. This enhancement allows the optimizer to investigate additional memory-related options, including activation evictions to off-chip memory (illustrated in Figure 5.2) and a new capability for weight fragmentation in convolutional layers, which offloads part of a layer's weights into the off-chip memory (illustrated in Figure 5.3). Together, these additions maximize memory efficiency and ensure that larger models and complex network structures can be accommodated without compromising performance, aiming to realise the full potential of off-chip memory resources, which have previously been underutilised.

The key contributions of this chapter are as follows:

- A Streaming CNN Accelerator for Efficient Memory Offloading: We propose a novel streaming CNN accelerator capable of partially offloading weights and activations to off-chip memory while minimising stalls in the computation pipeline. To further optimise off-chip memory transactions, we employ an encoding/decoding scheme, effectively reducing the off-chip memory bandwidth requirements.

- Subgraph-Based Partitioning Methodology: We present a subgraph-based partitioning approach that exploits FPGA reconfigurability to offer design trade-offs between latency and throughput. Building upon the partitioning strategies introduced in the previous chapter and integrating the option to offload data to off-chip memory, this approach is particularly effective in balancing the computational load across hierarchical and branched network structures.

- Design Space Exploration (DSE) Strategy: We introduce a DSE technique based on a greedy, iterative optimisation algorithm. This technique effectively explores the expanded design space, which incorporates subgraph-based partitioning and off-chip memory offloading options. By automating the exploration of memory (on-chip and off-chip) and partitioning configuration options, this approach ensures efficient resource utilization and high performance, particularly for models with complex branching and hierarchical structures.

Our proposed method advances the field of FPGA-based CNN acceleration by specifically focusing on optimising both on-chip and off-chip memory utilisation. This work paves the way for future research into more efficient and scalable hardware designs, especially as the demands for high-performance, AI systems continues to grow.

The contributions in this chapter are a collaborative effort, with specific responsibilities distributed as follows. The core mechanisms for leveraging off-chip memory—namely, activation eviction and weight fragmentation—originate from Z. Yu's prior work [126] and constitute his primary contributions to this study. These techniques are described in detail in Sections 5.3.1 and 5.3.2. Section 5.4.1 and Figures 5.1 and 5.5 are also Z. Yu's contribution. The remaining aspects of the work, are my own contribution.

5.2 Background

In a conventional streaming architecture, the majority of the memory footprint resides on-chip, where weights are stored in BRAMs and activations are buffered using FIFOs. To maximize the efficiency of on-chip memory usage, FINN [127] explored the overclocking of BRAMs, resulting in virtual memory ports to mitigate the mismatches between desired and available memory shapes, thereby enhancing BRAM utilization. Similarly, HPIPE [25] adopted a strategy of compressing sparse weight parameters using Run-Length Encoding (RLE), significantly reducing on-chip storage demands with minimal decoding overhead during operations. Addressing the constraints of limited on-chip memory, two recent works [128, 129] have proposed partitioning the computational workload across multiple FPGA devices, albeit sometimes at the cost of suboptimal utilization of individual FPGAs.

While single-engine architectures utilise tiling to overcome on-chip storage constraints, streaming architectures face greater challenges overcoming this limitation, especially with networks characterized by numerous parameters and lengthy interconnections necessitating substantial buffering. Our proposed approach extends certain concepts

introduced in [130, 126] and addresses these challenges through strategic management of both on-chip and off-chip memory. By selectively offloading weights and activations to off-chip memory, our method ensures the continuity of the computation pipeline in streaming architectures, mitigating potential stalls and enhancing overall performance. The proposed approach is orthogonal to previous methods, offering complementary strategies to further optimize FPGA-based CNN acceleration.

According to Table 5.1, numerous accelerator architectures emphasize CNN models targeting image classification tasks. These designs successfully handle widely-used networks such as VGG [131], GoogLeNet [35], ResNet [9], and MobileNet [38], evaluated on datasets like MNIST [132], CIFAR [133], and ImageNet [4]. Despite the focus on image classification, numerous other computer vision applications exist, including object detection, semantic segmentation, and video or volumetric data analysis. These tasks generally demand greater computational resources and involve more complex network architectures, posing significant new challenges for hardware accelerators.

The YOLO [125] family of models represents a set of highly accurate and efficient object detection algorithms that have become widely popular in computer vision. Utilizing a one-stage detection strategy, YOLO's neural network predicts bounding boxes and class probabilities in a single forward pass. The architecture includes multiple convolutional layers, detection layers, and a series of deep residual connections, with depths ranging from 20 to 60 layers. Angel-Eye [58] was among the pioneering efforts to support the YOLO model on FPGA, implementing an 8-bit quantized CNN backbone executed via a reconfigurable vector processor, while offloading pre- and post-processing tasks to a CPU. Nakahara *et al.* [134] took a similar approach, opting to binarize the YOLO model for improved hardware efficiency. More recent works, such as HPIPE [135] and SATAY [130], have focused on accelerating object detection tasks within streaming architectures, tackling challenges like optimizing high-performance Non-Maximum Suppression and HardSwish modules.

The UNet model [36] serves as a widely used solution for semantic segmentation tasks, featuring an architecture that incorporates both downsampling and upsampling operations to produce accurate segmentation masks. While the acceleration of convolutional layers has been extensively studied, designing hardware accelerators for UNet requires a focus on efficient deconvolution algorithms for upsampling. Liu *et al.* [136] developed parametrised engines for both convolution and deconvolution, utilizing shared input buffers to optimize performance. However, deploying UNet in a streaming architecture remains unexplored, primarily due to the model's long skip connections between downsampling and upsampling

layers—ranging from 7 to 40 layers—which necessitate deep on-chip buffers. Additionally, the substantial workload of 130.1 GMACs poses significant challenges for efficient mapping onto a streaming architecture.

Table 5.1 Comparison of our work with related works.

Works	Snowflake [124]	Angle-eye [58]	Brainwave [137]	[2]	[136]	Vitis AI [19]	DeepBurning [138]	FINN [23]	DeepBuilder [139]	HPIPE [25]	SMOF (Ours)
Architecture Style	Single Engine Architectures						Streaming Architectures				
Classification	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
Detection	✗	✓	✗	✗	✗	✓	✗	✗	✓	✓	✓
Segmentation	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓
2D CNN	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
3D CNN	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓

3D CNNs are often favoured for interpreting temporal information across video frames, due to their additional convolutional dimension. Shen *et al.* [2] proposed a template-based architecture that supports both 2D and 3D CNN models, leveraging the Winograd [140] algorithm to significantly reduce the number of multiplications required in convolutions, thereby lowering overall computational complexity. They also introduced an analytical approach to efficiently explore the design space and identify optimal tiling strategies. The fpgaHART framework, discussed in the previous chapter, evaluated a variety of 3D CNN models using streaming accelerator architectures. Nonetheless, neither of these methods could accommodate networks such as 3D UNet, which pose significant challenges due to their extensive on-chip buffer requirements caused by long skip connections and a heavy computational workload of 918.64 GMACs.

Existing streaming architecture accelerators have established a strong foundation for addressing the memory and computational challenges of various CNN architectures. However, the challenges posed by modern networks, such as YOLO, UNet, and X3D, highlight specific gaps that this chapter aims to address. For instance, the extensive on-chip buffering requirements of these models' long skip connections underscore the limitations of current methods in managing on-chip memory effectively. Our approach directly tackles these challenges by integrating off-chip memory management mechanisms into streaming architectures. The proposed activation eviction and weight fragmentation techniques mitigate the memory bottlenecks associated with long skip connections and large parameter sets. Furthermore, the extended partitioning strategy enables the efficient mapping of networks with hierarchical structures and extensive branching, as demonstrated in our evaluation of 2D and 3D CNNs in Tables 5.1 and 5.5. By addressing these challenges, this chapter bridges the gap between the limitations of existing methods and the growing demands of advanced computer vision tasks.

5.3 Methodology

In this section, we propose and formalize a threefold strategy to mitigate the on-chip memory bottlenecks commonly encountered in streaming architectures. This strategy is an extension of the toolflow presented in Chapter 4, providing additional flexibility and scalability through an expanded design space. To address these challenges, we introduce three innovative design choices: activation eviction, weight fragmentation, and an enhanced partition reconfiguration strategy that integrates off-chip memory offloading to manage computational loads more effectively in hierarchical and branched network structures. Activation eviction is designed to strategically offload and manage activations, reducing the on-chip memory resources and minimising the occurrence of stalls between the computational operations. Weight fragmentation optimizes the handling of large and dense weight matrices by breaking them into static and dynamic regions, allowing the weights in the dynamic regions to be loaded from the off-chip memory at runtime. Finally, partition reconfiguration enables the adjustment of the computational graph, allowing the comprehensive exploration of the trade-off between latency and throughput. Together, these strategies extend the toolflow's capabilities, enhancing its adaptability to diverse CNN model requirements and effectively managing both on-chip and off-chip memory.

5.3.1 Activation Eviction

In streaming design toolflows, the CNN workload is abstracted to a Directed Acyclic Graph (DAG) where nodes represent operations like convolution, pooling, and activation functions, while edges represent the data flows between these operations. One critical aspect in streaming architectures is to minimise the occurrence of stalls between the computational operations. In the case of sequential connections between nodes, buffers are inserted to address any instantaneous rate mismatches that might occur between consecutive operations, as the operations may produce or consume a burst of data to sustain their continuous execution.

Moreover, in instances where the DAG branches into multiple computational paths, the role of buffers becomes even more critical. Branch connections entail diverging paths where different computational branches may operate at varying throughputs or exhibit different processing latencies. Buffers placed at these branching points prevent a fast

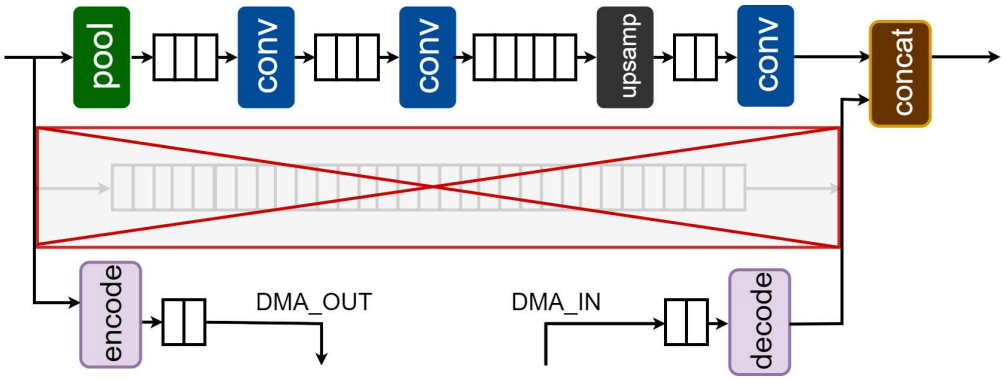


Fig. 5.2 Activation eviction. For the long skip connection, instead of being held in the on-chip buffer, the activation data is pushed to off-chip memory. We support lossless encoding schemes, such as RLE and Huffman, to save the off-chip bandwidth.

branch from overwhelming a slower branch. These buffers help maintain a balanced data flow, enabling synchronisation between branches and preventing potential deadlocks.

To reduce the on-chip memory resources required by implementing these buffers, we propose the eviction of a subset of activation data to the off-chip memory instead. Specifically, the original deep buffer with the depth of d_b, introduced in "Split After Branch Merging Point" in Section 4.3.2, is replaced with the circuit shown in Figure 5.2, which only contains two small FIFOs to only hold the data chunk enough for a DMA burst, and the remaining activation data are pushed off-chip. At the host side, pointers are used to manage the head and tail positions of the evicted activation data.

An important question to answer is at which edge should the eviction be applied. A considerable number of design choices are available, as we can choose to evict the edge of any two connected nodes. Let us start with establishing the trade-off between on-chip and off-chip memory resources for transferring activations in the streaming architecture.

If we denote the total depth of the two small FIFOs after eviction as d'_b and compare it with the original buffer depth of d_b, the saving in buffer depth will be:

$$\Delta d_b = d_b - d'_b, \quad s.t. \quad d_b > max(d'_b, t_{db}) \tag{5.1}$$

In this equation, we also apply an extra constraint that the original buffer depth d_b must be larger than the delay of DMA transfers t_{db}. Otherwise, the eviction will cause extra stalls in the computational pipeline, downgrading the overall performance. The width of the buffer is determined by the activation word length multiplied by the parallelism in the input channels, $wordlength_{act} \times c^{in}$.

On the other hand, the extra off-chip bandwidth required, including both writing and reading, after the eviction can be calculated as,

$$\Delta BW_b = r * \bar{c} * (1 + \alpha) \tag{5.2}$$

where the average data rate is represented as r (words/cycle). Let us use $\bar{c}$ to denote the average compression ratio for DMA transfers, as activations can be encoded losslessly using formats such as Run-Length Encoding (RLE) or Huffman.

Note that the compression ratio for activation data actually varies on different input images. At design time, we use the average ratio to estimate the bandwidth requirement, and we will analyse the deviation of this estimation and its performance impact at the end of Section 5.5.3.

In Equation (5.2), α is a penalty factor for the reading bandwidth, with a value that is always larger than or equal to one. The penalty factor α becomes greater than one when data ordering when reading and writing to the buffer differs. This scenario typically occurs when the parallelism factors of reading and writing on the buffer are mismatched, necessitating a reordering of the data. Such reordering transforms the off-chip reading into random access, which incurs a bandwidth penalty.

Overall, Equations (5.1) and (5.2) are utilised to quantify the trade-off between on-chip and off-chip memory resources for transferring activations in the streaming architecture. The relationship is captured by the ratio $\Lambda * \Delta d_b / \Delta BW_b$, as further elaborated in Section 5.4.2, where the ratio of this trade-off is used to determine where the activations should be evicted.

5.3.2 Weight Fragmentation

As a common practice in streaming architectures, convolution operations require the corresponding weight parameters completely stored on-chip. Each operation is assigned its own dedicated memory block so that these operations can be parallelised and pipelined. As a way to reduce on-chip memory requirements, a memory fragmentation scheme is introduced, where the original weight memory of depth d_w, is now fragmented into static and dynamic regions, as Figure 5.3 shows.

Weights in the static regions are fixed and read-only, and these static regions are concatenated together to be stored in a continuous, physical on-chip memory space; while weights in the dynamic regions need to be loaded from off-chip memory at runtime, sharing

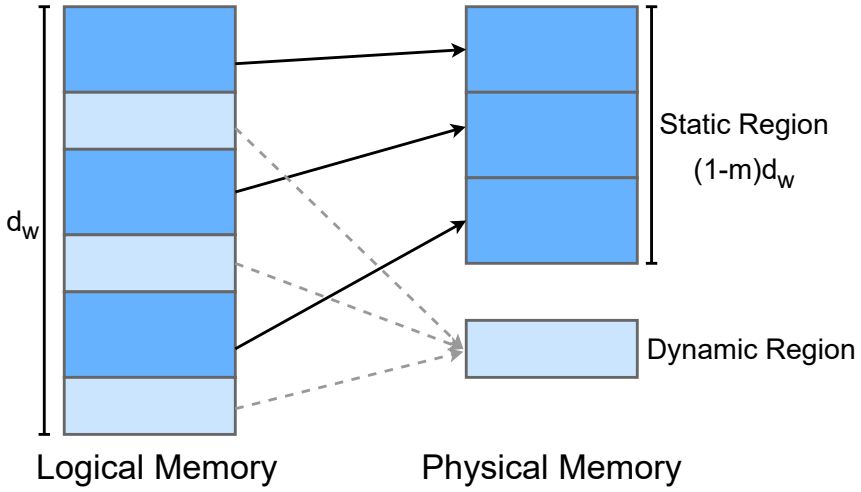


Fig. 5.3 Weight fragmentation. The original weights with depth of d_w is fragmented into static and dynamic regions. Weights in the dynamic regions are loaded from the off-chip memory at runtime, sharing the same piece of physical memory space in a time-multiplexed manner. The ratio of dynamic regions is denoted as m.

the same physical memory space in a time-multiplexed manner. The computational pipeline iteratively consumes the weights from either the static or dynamic regions.

For each convolution operation, the ratio of its dynamic regions $m \in [0, 1]$ is a tunable design parameter, where $m = 0$ refers to the situation that all weights are static and no off-chip loading is required, while $m = 1$ represents moving everything to off-chip instead. Given this ratio, we can represent the saved on-chip memory depth as

$$\Delta d_w = m * d_w, \tag{5.3}$$

where d_w denotes the original memory depth before fragmentation and the width of the buffer is determined by the weight word length multiplied by the parallelism in the input and output channels as well as the parallelism in the dot product calculation, $wordlength_{weight} \times c^{in} \cdot c^{out} \cdot f$.

Meanwhile, the required off-chip memory bandwidth to load the dynamic regions can be represented using the following equation:

$$\Delta BW_w = m * r * \grave{c}, \tag{5.4}$$

where the computational pipeline consumes weights at the rate of r (words/cycle), and similar to activations, weights can also be encoded with compression ratio $\grave{c}$, reducing the required bandwidth, but, in contrast to activations, this ratio is known at compile-time for weights.

The decision of which convolutional layers should have their weights fragmented, and by how much (i.e., the value of the fragmentation ratio m), is a critical part of the automated optimisation process. This is not a manual choice but is determined during the Design Space Exploration (DSE) stage of the toolflow. More details in how this ratio is tuned are provided in Section 5.4.2.

5.3.3 Partition Reconfiguration

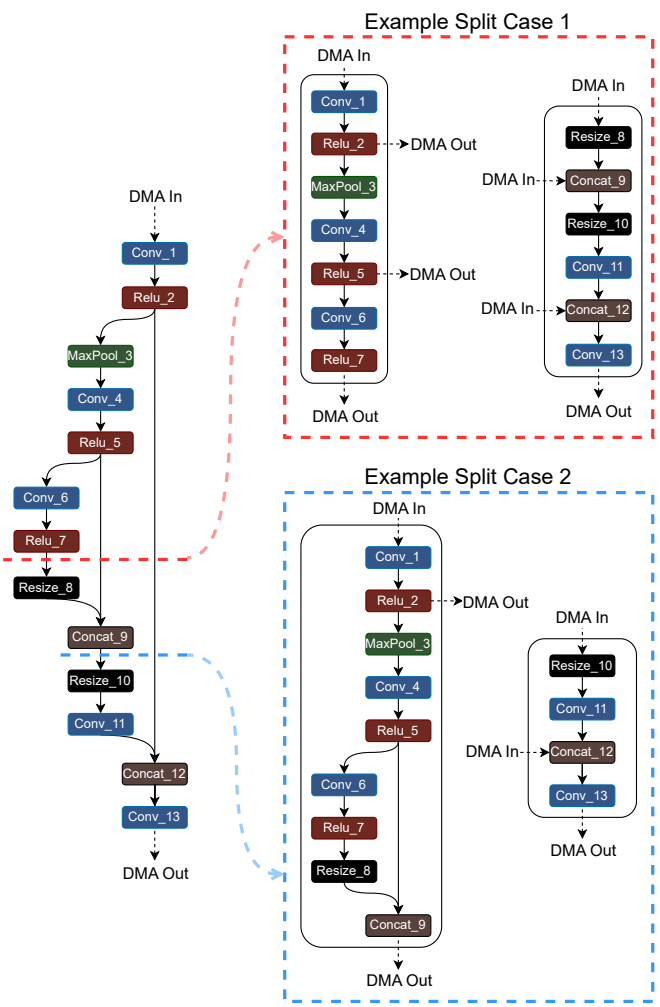
The runtime reconfiguration capability of FPGAs enables the CNN workload to be divided into multiple sequentially processed partitions, for enhanced resource utilization and performance [24]. As detailed in Section 3.4.2, the SDFG G of the target CNN is divided into $\mathcal{P}$ partitions, each processing the workload of a certain part of the model. Based on the batch size B, the initiation interval II_{max}, and the pipeline depth $D_{\mathcal{P}_i}$ as they have been described in Chapter 3, we are able to infer the throughput (Equation 3.6) of each partition and of the complete model.

Figure 5.4 illustrates the techniques employed to modify the computation graph. Graph partitioning, as thoroughly discussed in Section 4.3.2, is enabled, along with fragmentation or eviction at any arbitrary point, provided the following three constraints are satisfied:

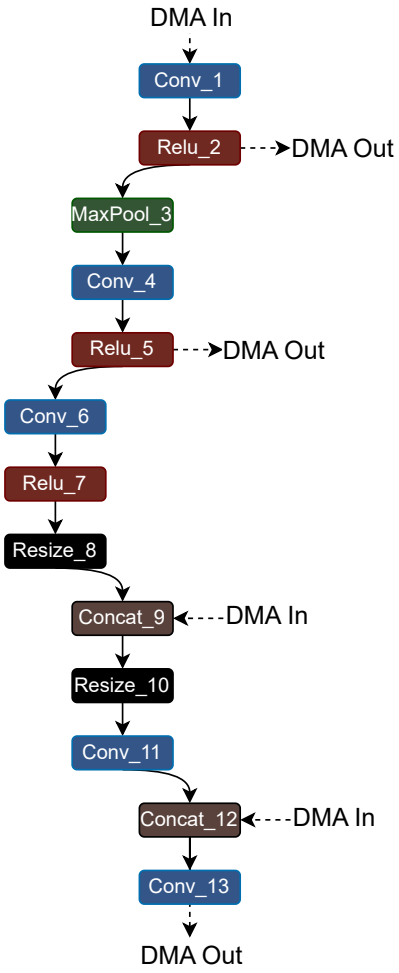
① *on-chip resources*: Each partition's utilisation of on-chip FPGA resources, including Digital Signal Processing Blocks (DSP), Look-Up Tables (LUT), Flip-Flops (FF), and Block RAM (BRAM), must adhere to the limits imposed by the FPGA device.

② *compute dependency*: Partitions are scheduled to execute sequentially on a single FPGA device in a time-multiplexed manner. Therefore, it is crucial to maintain the correct sequence of computations. For any node within a partition, all of its producers must either reside within the same partition or have already been executed in a previously scheduled partition.

③ *off-chip bandwidth*: The off-chip bandwidth utilisation of each partition, comprising the partition's inputs and outputs, as well as the overhead related to activation eviction and weight fragmentation, must comply with the bandwidth limitations of the FPGA device to prevent bottlenecks or performance degradation.



(a) Graph splitting strategies.



(b) Graph off-chip streaming strategy.

Fig. 5.4 Graph manipulation strategies. Figure 5.4a showcases the graph partitioning strategy into multiple partitions for different reconfiguration points. In Figure 5.4b the off-chip streaming strategy is presented, utilising the activation eviction method. By streaming the long skip connections to off-chip and reading back from it on merge points the graph can fit into the device without needing multiple partitions and hence device reconfiguration.

While the first two constraints—on-chip resource utilization and compute dependency—are hard constraints that must be strictly satisfied to ensure correct functionality, the third constraint, off-chip bandwidth, is a soft constraint. Exceeding the available off-chip bandwidth does not make a design invalid but can lead to reduced performance due to stalls caused by communication delays with the off-chip memory. This distinction is critical to understanding the trade-offs during the partitioning process, where designs violating bandwidth constraints may still be viable, albeit at a cost to performance as discussed in detail in Section 5.5.3 and showcased in **??**.

Graph partitioning or fragmentation/eviction are not exclusive, and they can be combined together for the optimal performance and resource trade-off, leaving a large design space to explore. We will elaborate our exploration algorithm in Section 5.4.2.

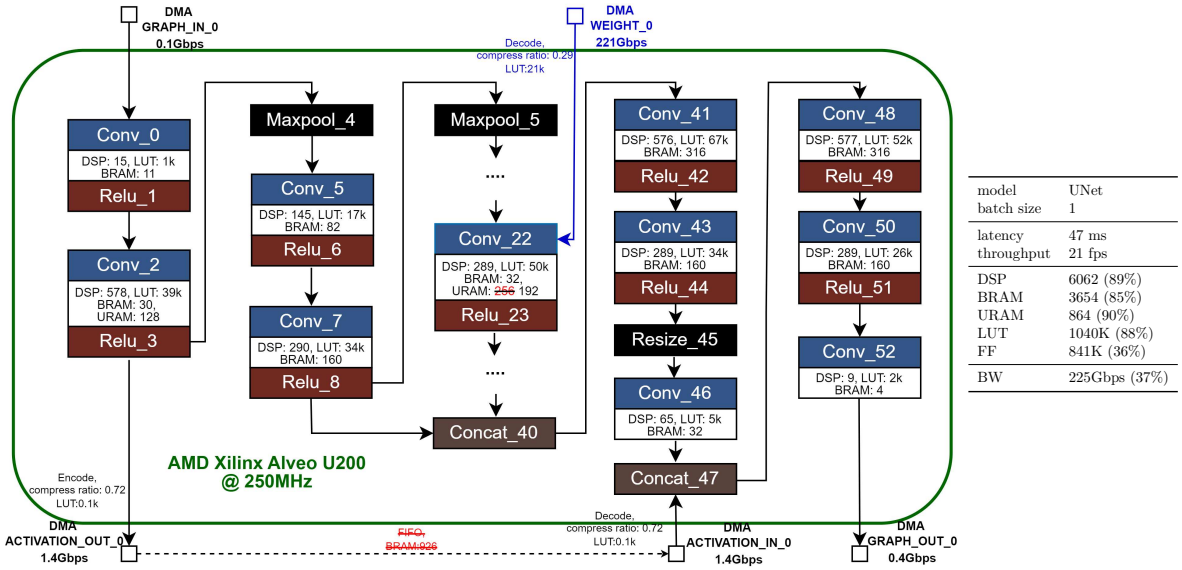
5.4 Implementation

In this section, we present an overview of the implementation details of our proposed system architecture, highlighting the key components and methodologies that contribute to its efficiency. We discuss the specifics of our Design Space Exploration (DSE) optimization strategy and introduce a revised method for calculating and modelling the pipeline depth of the computation graph, which is essential for understanding and predicting the performance of the streaming architecture on models with excessive and deep skip connections.

5.4.1 System Architecture

We incorporate the proposed approach with an open-source toolflow, fpgaConvNet [24]. We use fpgaConvNet as a code generator that translates a PyTorch description of the CNN to the RTL files, and stitches those files together to build the computational pipeline.

As with other existing streaming architectures, the computation pipeline generated by the original fpgaConvNet toolflow restricts most data access to on-chip memories. When incorporating activation eviction, we keep the hardware of the computation nodes unchanged and modify their connections as in Figure 5.2 which exposes two additional DMA ports. In the case of weight fragmentation, the original on-chip memory block is reduced as in Figure 5.3 and the shared buffer for the dynamic regions is connected to one read-only DMA port.



model	UNet
batch size	1
latency	47 ms
throughput	21 fps
DSP	6062 (89%)
BRAM	3654 (85%)
URAM	864 (90%)
LUT	1040K (88%)
FF	841K (36%)
BW	225Gbps (37%)

Fig. 5.5 A streaming accelerator design example with the proposed memory optimisation methodology, where the weights of the layer `Conv_22` are partially offloaded to off-chip, and the long skip connection between `Relu_3` and `Concat_47` is also evicted to off-chip.

The encode/decode modules are attached to the DMA ports if requested, with support for both RLE and Huffman schemes. These schemes were chosen for their widespread use, the availability of FPGA implementations, as well as their relatively low resource overheads, as demonstrated in Table 5.4. A more detailed discussion on the impact of the two encoding schemes can be found in Section 5.5.3. If the total number of required DMA ports is more than the available DDR banks, time-multiplexing is applied.

Figure 5.5 presents a UNet design example with the layer-wise computation pipeline and the off-chip memory connections visualized. The input images are fed into the pipeline through `Conv_0`, and the segmentation masks are generated from `Conv_52` which is the last layer. As the figure shows, weight fragmentation is applied to `Conv_22`, which reduces the URAM usage of this convolutional layer from 256 to 192, saving 18 MB on-chip memory at a cost of 21K LUTs for decoding and 221 Gbps bandwidth. In addition, activation eviction is applied between `Relu_3` and `Concat_47` which further saves 926 BRAMs, equivalent to 16 MB on-chip memory, at a cost of 2.8 Gbps and negligible LUT overhead.

5.4.2 DSE and Optimisation

To obtain a design like Figure 5.5, there are the following design choices that need to be decided:

- *how many partitions should we have after partitioning?*
- *how should on-chip resources be allocated along the computation pipeline?*
- *where should activation eviction be applied?*
- *where should weight fragmentation be applied and how should the fragmentation ratio be decided?*

The above four factors can be formulated into an optimisation problem, where the objective is to maximize the system's throughput $\mathcal{T}(G)$ (Equation 3.6) or minimize the latency $\mathcal{L}(G)$ (Equation 3.5) for a given model graph G, subject to the FPGA device constraints of on-chip resources $\mathcal{R}_{FPGA}$ and off-chip memory bandwidth BW. When there is only a single partition, these objectives align, as maximizing throughput inherently minimizes latency. However, with multiple partitions, trade-offs arise between these objectives, hence we define two different optimisation objectives, one for minimizing latency and another one for maximizing throughput:

$$\min_{\mathcal{V}_n} \mathcal{L}(G), \quad s.t. \quad \sum_n \mathcal{R}(n) \leq \mathcal{R}_{FPGA}, \sum_n BW(n) \leq BW \quad \forall n \in \mathcal{P}, \forall \mathcal{P} \in G \tag{5.5}$$

$$\min_{\mathcal{V}_n}(-\mathcal{T}(G)), \quad s.t. \quad \sum_n \mathcal{R}(n) \leq \mathcal{R}_{FPGA}, \sum_n BW(n) \leq BW \quad \forall n \in \mathcal{P}, \forall \mathcal{P} \in G \tag{5.6}$$

where G, $\mathcal{P}$, n represent the graph, partitions, and nodes respectively. In the above formulation, we use a $-\mathcal{T}(G)$ to convert throughput into a minimization objective, ensuring that higher throughput contributes positively to the optimization goal.

For each individual node, its design options are represented by the vector $\mathcal{V}_n$, which consists of a set of tunable compile-time optimization parameters including:

- binary flags s_i, s_o indicating if the current node is the input or output of a partition,

- the operation parallelism p,
- binary flags a_i, a_o indicating whether its input or output activations are evicted off-chip,
- weight fragmentation ratio m.

Determining the values of the vector $\mathcal{V}_n$ is part of the Design Space Exploration (DSE) process. fpgaConvNet proposed a greedy algorithm [141] to identify the optimal operation parallelism p, and in this work, we extend this algorithm to cover the exploration of the additional design choices that are related to off-chip memory. Specifically, our DSE algorithm includes the following passes:

① *resource-minimal initialisation*: we start the DSE with the configuration that has minimal resource requirement. Specifically, we can partition the target DAG into as many partitions as possible, and inside each partition, each node is also set as it's minimal parallelism level. Users can also constrain the partitioning positions by specifying only the certain types of operation that can become the partition inputs/outputs, with the trade-off between DSE time and quality.

② *compute parallelism allocation*: for the computation pipeline in a partition, it is preferable to increase the parallelism of the slowest operation so that the pipeline initial interval II_{max} (Equation 3.4) can be reduced. Whenever the parallelism of the slowest operation cannot be further increased, we move on to other operations and allocate compute resource there if it helps reduce pipeline depth $D_{\mathcal{P}_i}$.

③ *on-chip memory allocation*: on an AMD Xilinx FPGA device, BRAMs and LUTRAMs are typically available to be the on-chip memory, as well as URAMs on certain devices. These on-chip memory resources are configured to store weights and activations as required. Our algorithm balances the utilisation ratios among resource types and also optimises utilisation efficiency considering that these on-chip memory resources can only be configured with specific width and depth ratios.

④ *off-chip bandwidth allocation*: this pass decides the binary flags a_i, a_o for activation eviction and also the weight fragmentation ratio m. All the nodes in the partition, are sorted by the heuristic metric $\Lambda * \Delta d / \Delta BW$, in which Λ denotes the data word length, while Δd and ΔBW denote the change in the on-chip memory depth (Δd_b from Equation (5.1) and Δd_w from Equation (5.3)) and off-chip memory bandwidth (ΔBW_b from Equation (5.2) and ΔBW_w from Equation (5.4)) respectively. A large quotient indicates significant savings for the on-chip memory with a relatively small cost

in off-chip memory bandwidth, so the off-chip memory bandwidth will be allocated for that node with priority.

⑤ *partition merging*: since we start with splitting the DAG into as many partitions as possible, this pass focuses on the merging of the partitions. When we merge two partitions, the total device reconfiguration time is reduced since there are fewer partitions, however, the merged partition usually has less parallelism due to the device resource constraint, or in an extreme case, the merged partition might not fit on the target FPGA device at all. Therefore, this pass considers the performance estimation before and after any potential merging and takes the decision based on which configuration yields better performance whilst not exciding the FPGA available resources.

The above passes are combined in an iterative manner, as Algorithm 3 shows, allowing the algorithm to refine the design configuration incrementally. Eventually, the DSE process identifies the optimal design configuration which is used to generate the RTL code.

Algorithm 3 Design Space Exploration

Require: G, B ▷ CNN graph, batch size
Require: $\mathcal{R}_{FPGA}, BW, t_{reconfig}$ ▷ on-chip rsc, bandwidth, reconf time
  $\mathcal{P} \leftarrow$ initialize(G) ① ▷ resource minimal partitions
  for $\mathcal{P} \in G$ **do**
    while $\sum_n \mathcal{R}(n) \leq \mathcal{R}_{FPGA}$ and $\sum_n BW(n) \leq BW$ **do**
      alloc_parallel($\mathcal{P}$, $\mathcal{R}_{FPGA}$) ②
      alloc_on_chip($\mathcal{P}$, $\mathcal{R}_{FPGA}$) ③
      alloc_off_chip($\mathcal{P}$, BW) ④
  merge_partition($\mathcal{P}_i, \mathcal{P}_j$) $\forall \mathcal{P}_i, \mathcal{P}_j \in G$ ⑤

The proposed algorithm leverages its greedy nature, combined with carefully selected heuristics, to iteratively optimize the trade-offs between on-chip memory allocation and off-chip bandwidth utilization. The systematic partitioning and merging of model sub-graphs enables a fine-grained exploration of the design space, ensuring that resource constraints are respected while maximizing performance by exploring trade-offs between latency and throughput. This approach strikes a practical balance between computational efficiency and memory utilization, addressing the scalability challenges posed by modern CNN model architectures consisting of complex branching and hierarchical structures.

5.4.3 Pipeline Depth Estimation

Given the importance of pipeline depth in influencing both the overall performance of a design for a computation graph G and its impact on memory resources, we propose a revised approach to estimate pipeline depth, expanding upon the buffering introduced in Chapters 3 and 4. The accurate estimation of pipeline depth is critical because it directly affects the calculation of the buffering needed in branches, which has a significant impact on the on-chip memory utilisation. During DSE, where we search for optimal designs, an overestimation of pipeline depth may lead to discarding designs that are practically feasible, while underestimation can result in designs with deadlocks caused by insufficient buffering in the branches. To begin, the pipeline depth for each node n, of a graph G, is closely related to its initiation rate $r^{st}(n)$, which determines the rate of a node during the initial stages of its execution within the pipeline. Notably, this initiation rate differs from the rate sustained during the remaining execution of the layer. This rate is calculated based on the interval preceding the node, expressed as:

$$Interval_{prev}(n) = \max(\lambda_a + D_a) \quad \forall a \in ancestors(n)$$

where $ancestors(n)$ denotes all nodes directly preceding node n in G, λ_n represents the latency for node n, and D_n indicates its pipeline depth. This latency estimation λ_n is derived using performance models adapted from fpgaConvNet [24], specific to each layer type.

The initiation rate $r^{st}(n)$ is then defined as follows:

$$r^{st}(n) = \begin{cases} r_n^{in} & ancestors(n) = \emptyset \\ \frac{S_n^{in}}{Interval_{prev}(n)} & otherwise \end{cases}$$

where r_n^{in} is the input rate or consumption rate of node n, and S_n^{in} denotes the size of its input feature map. This calculation helps clarify the variation in initiation rate within specific pipeline regions, as illustrated in Figure 5.6. The waveform shows an annotated segment of a synthesised design, depicting different computation regions in which the rate of a given layer varies based on the region it resides. Notably, the initiation rate r_n^{in}, is observed in the pipeline depth region, contrasting with the input rate r_n^{in} at the remainder of the execution.

The cumulative delay for a node n within the graph G is subsequently determined using:

$$Delay(G,n) = \sum_{i=0}^{max(P_G(N_G^{in},n))} \frac{D_n}{r^{st}(i)}$$

where the function $P_G(src, trg)$ enumerates all the valid paths in the graph G originating from the source node (src) and terminating at the target node (trg). This refined delay estimation for each node captures the variable initiation rates across different regions of the computation.

In conclusion, the equation below presents a refined model for the fpgaConvNet's pipeline depth estimation, as was previously described in Equation 3.4 with the term D. The updated estimation accounts for the distinct rates present in each layer's computation. This revised version aligns more closely with the actual hardware implementation, offering a more accurate representation and estimation of the model's performance estimates.

$$D_{pG} = \max_{n \in G}(Delay(G,n))$$

With the incorporation of this updated pipeline depth estimation model, and consequently, the revised latency model (Equation 3.4), we have ensured deviation level of approximately 12% on our proposed designs. The extent of this deviation may vary slightly depending on the structure of each model.

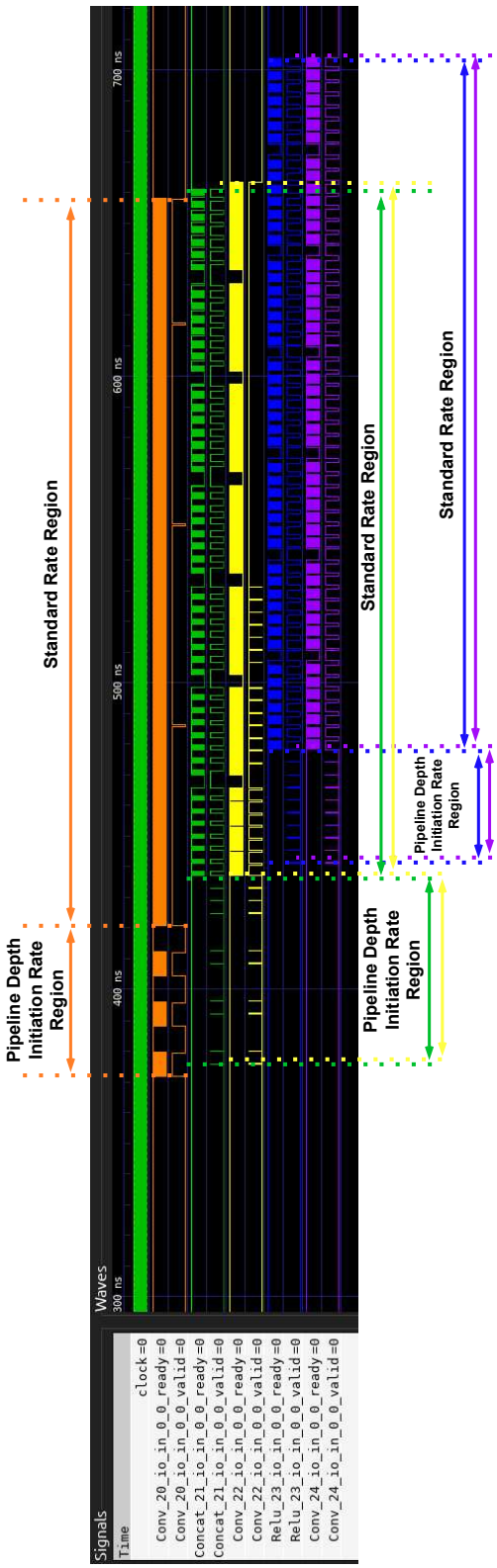


Fig. 5.6 A segment of the design's waveform illustrates the distinction between the initiation rate r^{st}, which is applicable to the pipeline depth region of the layers, and the input rate r^{in}, evident for the remainder of the layer's execution.

5.5 Performance Evaluation and Results

This section presents the experimental results of our proposed method's auto-generated accelerators validated for a range of CNN models (both 2D and 3D) targeting a range of AMD/Xilinx platforms, including the ZCU102 board (Zynq UltraScale+ XCZU9EG), the Alveo U200 accelerator card (Virtex UltraScale+ XCU200), and the Virtex UltraScale+ VCU1525 (XCVU9P) and VCU118 (XCVU9P) boards. The models included in the evaluation are listed in Table 5.2. The selected models have demonstrated exceptional performance across diverse computer vision tasks such as image segmentation (UNet [36]), object detection (YOLOv8 [125, 142]), 3D Volumetric Segmentation (UNet3D [143]), as well as action recognition (X3D [6]). Aside from supporting a range of computer vision applications and incorporating both 2D and 3D CNNs, the selected models display a wide range of workload and network characteristics, with the unifying feature of integrating long skip connections into their architectures.

Table 5.2 Characteristics of the evaluated CNN models.

Model	Dataset	MACs (G)	Params (M)	# Layers	# Conv Layers	Input Dims	Accuracy (%)
Yolov8n	COCO$^{\pm}$[144]	4.37	3.16	115	63	(3, 640, 640)	37.10 (Float32) 37.10 (W16A16) 29.60 (W8A8) 35.10 (BFP8)
UNet	CamVid‡[5]	130.12	28.96	53	23	(3, 368, 480)	71.67 (Float32) 71.67 (W16A16) 60.62 (W8A8) 71.75 (BFP8)
UNet3D	BraTS2020$^{\mp}$[145]	918.64	5.65	52	19	(4, 155, 240, 240)	85.34 (Float32) 85.23 (W16A16) 1.15 (W8A8) 85.34 (BFP8)
X3D-M	UCF101†[1]	6.97	3.82	396	115	(3, 16, 256, 256)	96.40 (Float32) 96.40 (W16A16) 0.81 (W8A8) 96.29 (BFP8)

$^{\pm}$ Accuracy reported in mAP50-95. ‡ Accuracy reported in mIOU. $^{\mp}$ Accuracy reported in Dice Score.
† Accuracy reported in Top-1 Acc.

5.5.1 Baseline Setup

In this section, we briefly describe the baseline configuration for the ablation study of the next sections. We utilise quantisation to further enhance the performance of our designs and to be aligned with best practices in FPGA-based accelerator development. Weights are quantized depending on minimum and maximum values within each channel of each layer, using a channel-wise block floating-point format with an 8-bit wordlength.

This is an efficient quantization scheme where all weights belonging to a single channel are mapped to 8-bit integers using a shared floating-point scale factor and zero-point. This approach effectively preserves the unique dynamic range of each channel, thereby minimizing the accuracy loss typically associated with using lower-precision arithmetic. Without additional fine-tuning, this process is applied directly to pre-trained weights. Activations are likewise quantized with an 8-bit fixed wordlength, based on a calibration set of data to capture their range. The quantisation approach has a modest accuracy drop, as shown in Table 5.2, while providing significant improvements in resource utilisation and model performance. Moreover, we leverage the device's URAM, whenever it is supported, to balance the utilisation ratios between BRAM and URAM, thereby optimising overall utilisation efficiency as detailed in Section 5.4.1.

The reconfiguration of the device and partitioning of the models provide further enhancements to the baseline configuration. As shown in Table 5.3, properly partitioning the models across the available resources on the FPGA allows for larger models to be deployed efficiently. This partitioning strategy, supplementary to the quantisation and URAM utilisation techniques, enables the mapping of models that would otherwise exceed the on-chip memory capacity. From this point onward, the term "baseline" will refer to the designs incorporating all the features mentioned in this section, including quantisation, URAM utilisation, and model partitioning.

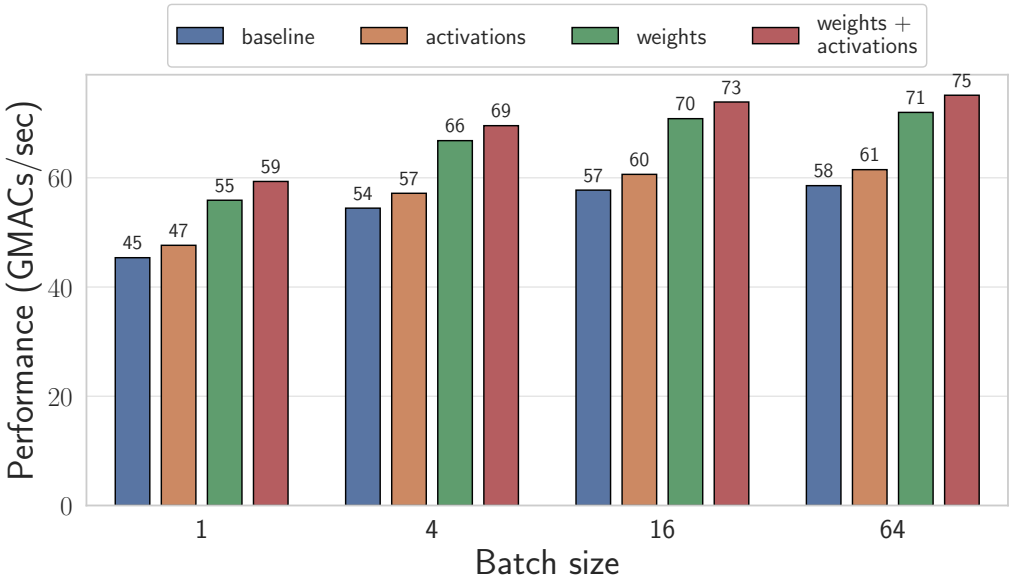
Table 5.3 Breakdown and impact of model partitioning with device reconfiguration on UNet3D designs targeting different batch sizes.

Batch Size	1	4	16	64
Number of partitions	4	5	6	6
Batch Latency†(sec)	0.77	2.76	9.54	36.69
Batch Compute Time†(sec)	0.53	2.43	9.13	36.28
Batch Reconfig Time†(sec)	0.24	0.33	0.41	0.41
Reconfig Contribution on Batch Latency†(%)	31.16	11.95	4.29	1.11

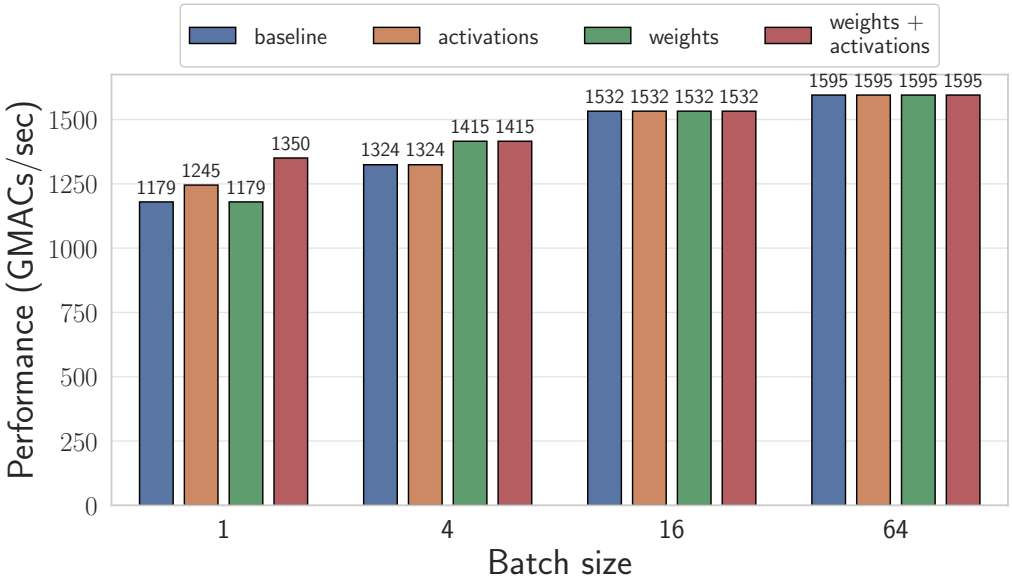
† Cumulative value summed across all partitions.

The primary focus of the ablation study is on two models: Unet, a 2D model, and Unet3D, a 3D model. However, it is essential to note that the conclusions derived from this comprehensive analysis are applicable and hold true for the other models evaluated in the final section.

5.5.2 Off-Chip Streaming



(a) UNet performance under different off-chip streaming strategies



(b) UNet3D performance under different off-chip streaming strategies

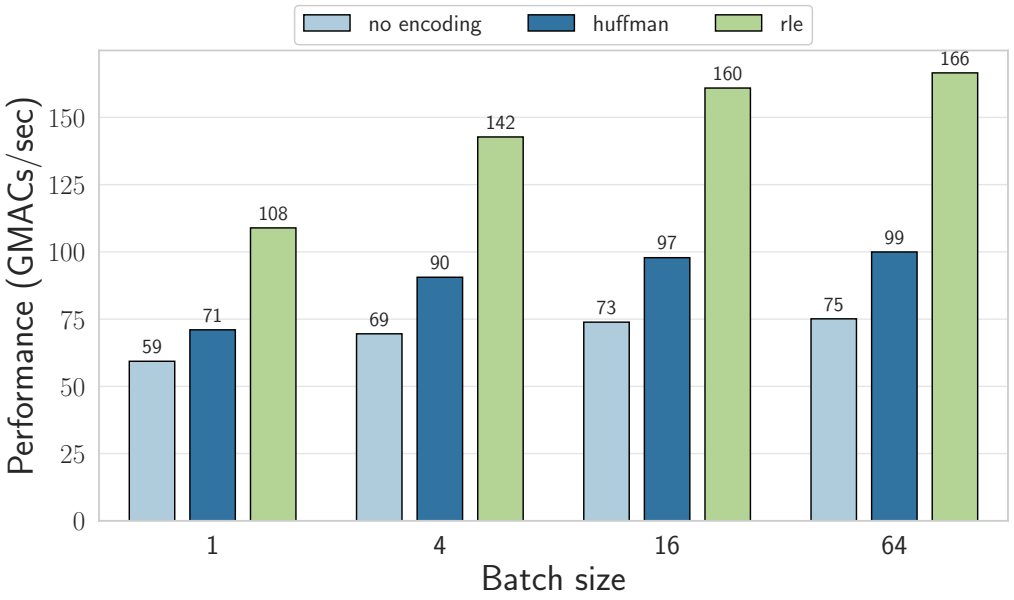
Fig. 5.7 Evaluation of the UNet and UNet3D under four different steaming strategies: *i*) No off-chip streaming allowed (blue), *ii*) Only activations off-chip streaming allowed (orange), *iii*) Only weights off-chip streaming allowed (green), *iv*) Both weights and activations off-chip streaming (red).

In Section 5.3, we discussed both activation eviction and weight fragmentation to overcome the on-chip memory bottleneck. To better understand their individual contribution to overall performance, we conduct an ablation study in Figure 5.7, and have the following observations:

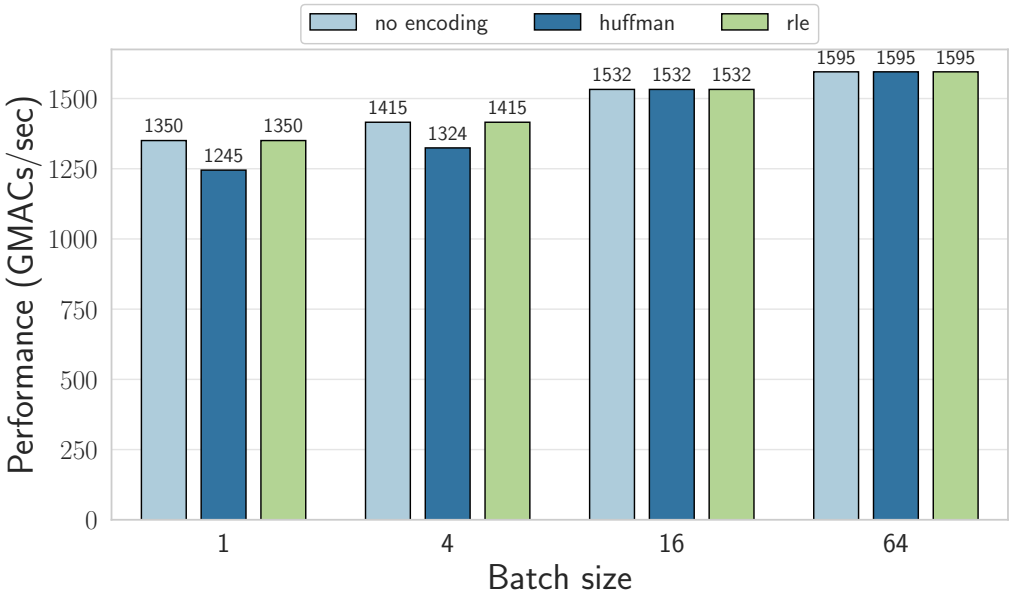
- The blue bars are the baseline designs, as described in Section 5.5.1, which have no support for activation eviction and weight fragmentation at all.
- When enabling activation eviction only (orange bars), compared with the baseline (blue bars), there is a small performance improvement for the UNet model of $\sim 5\%$. In the case of UNet3D with a batch size of 1, there is a notable improvement in performance, increasing from 1179 to 1245 MACs/sec.
- Moreover, in the case of UNet, there are performance improvements over both the previous cases when using weight fragmentation only (green bars). The same does not apply for the UNet3D case where weight fragmentation only performs the same as the baseline approach.
- For both the cases of UNet and UNet3D (for batch size 1) the use of both methodologies (red bars) demonstrates significant improvement in performance up to 1.3× for UNet and 1.1× for UNet3D. These results showcase the potential performance gains when activation eviction and weight fragmentation are enabled under different model workloads and FPGA devices.
- We also found the benefits of activation eviction and weight fragmentation are more obvious under smaller batch sizes. For a relatively large batch size such as 64, our DSE algorithm tends to create more partitions instead of using activation eviction and weight fragmentation methodologies. This is due to the amortization of the reconfiguration cost as the batch size increases, as shown in Equation 3.5.

5.5.3 Off-Chip Streaming Compression

To efficiently utilise off-chip memory bandwidth, we transfer data in encoded formats such as Huffman and RLE. The performance impact of these encoding schemes is shown in Figure 5.8 and the resource overhead is presented in Table 5.4. In terms of the resource overhead, it accounts for the encoding and decoding modules per data stream where each stream is originally is 8-bits wide. When applied to multiple parallel data streams, the overhead needs to be multiplied by the number of data streams as well.



(a) UNet performance under different compression strategies



(b) UNet3D performance under different compression strategies

Fig. 5.8 Evaluation of the UNet and UNet3D under three compression strategies for off-chip streaming. The off-chip streaming configuration is fixed to *weights & activations streaming*. The different colours denote the encoding scheme adopted: *No encoding*, *Huffman encoding*, and *Run-Length Encoding (RLE)*.

In order to evaluate the proposed encoding schemes, we apply the off-chip streaming configuration outlined in the preceding section, which enables weight fragmentation and

	Huffman IP		RLE IP	
	Encode	Decode	Encode	Decode
LUT	11	400	26	36
FF	7	81	10	11

Table 5.4 Resource overheads for using Huffman and RLE schemes. The overhead is applied per data stream containing 8-bit data. For compressing parallel data streams, the overhead will be multiplied by the number of data streams as well.

activation eviction. For UNet (Figure 5.8a), we observe the RLE scheme is the best choice, providing up to 2.21× MACs/sec compared with the non-encoding case, and 1.68× MACs/sec compared with the Huffman case. However, in the case of UNet3D (Figure 5.8b), applying these encoding schemes does not provide any further performance gain, which is due to the fact that the design becomes LUT-bounded instead of BRAM/bandwidth-bounded. In fact, when using Huffman encoding for UNet3D, the performance actually drops slightly due to the resource overhead for encoding and decoding, making the LUT-bounded situation even worse in this case. Overall, we found offloading weights and activations to the off-chip memory is beneficial to performance, but whether to further apply any encoding scheme may vary case by case.

5.5.4 Comparison with related FPGA works

Table 5.5 presents a thorough comparison of the accelerator designs produced by our method with those from previous studies. The evaluation includes a range of 2D and 3D Convolutional Neural Network (CNN) models on several FPGA platforms. Our solution has a significant advantage due to its adaptability. Our proposed solution has the ability to produce a broad spectrum of accelerators tailored to the characteristics of each one of the four major computer vision applications that we target: image segmentation, 3D segmentation, object recognition, action recognition. This adaptability extends across both 2D and 3D spatial domains, showcasing the adaptability of our approach in addressing a wide spectrum of applications.

Image Segmentation

In regards to image segmentation using UNet, prior studies improved and optimised the model architecture, resulting in a reduction of the overall number of operations of the original model. Variations in input size also affect the total MAC count. For this reason,

we provide three metrics: *fps, GOP/s, and GOP/s/DSP*, to account for variations in MACs as well as device diversity.

In comparison to [69], our UNet solution on VCU1525 demonstrates similar fps performance. However, our design showcases a notable 10.65× improvement on throughput expressed as the GOP/s. Due to the ZCU102's limited resources and the large workload of our UNet model (130 MACs versus 5.6 in [19], a 23.3× increase in model workload), our design becomes compute bounded on this board, resulting in no performance improvement over [19]. However, targeting a larger device with more resources, such as the U200, our design demonstrates a 2× improvement in GOP/s/DSP over [19]. This metric, as explained earlier, is device-agnostic as it normalises over the number of DSPs. We likewise achieve a 2.8× improvement over [136] under the same metric and board. Despite achieving 1.7× greater GOP/s performance than [146], our solution achieves half the GOP/s/DSP, which is likely owing to DSP implementation variations between Intel and AMD devices.

3D Segmentation

No prior work has been identified for the 3D segmentation task using UNet3D. Our proposed design achieved a performance of 1595 GOP/s targeting the U200 FPGA board.

Object Detection

In object recognition with YOLOv8, our proposed method achieves a 23× fps improvement over [130]. However, it exhibits a slightly lower GOP/s/DSP performance of 0.16 compared to the 0.18 reported in [130].

Action Recognition

In comparing the results for the X3D-M model, this chapter demonstrates a substantial performance improvement over the corresponding design in Chapter 4. On the same ZCU102 board, the throughput achieved in Chapter 4 is 13.44 clips/sec, whereas in this chapter, we achieved a throughput of 27.08 clips/sec—an improvement of 2.01×. This significant increase underscores the impact of the methodologies introduced in this Chapter, particularly the strategies for optimized off-chip memory utilization and the balancing between off-chip and on-chip memory resources. Additionally, the enhanced DSE contributed to more efficient selection of design points, further boosting throughput. Another factor potentially contributing to the performance gains is the adoption of 8-bit

block floating point (BFP8) precision, in contrast to the 16-bit fixed-point (W16A16) precision used previously. The impact of BFP8 is multifaceted. Firstly, there is a reduction in DSP and BRAM utilization, as demonstrated in Table 5.2. While the BRAM usage is comparable to that of the Chapter 4 design, the reduced number of partitions indicate that each partition can now accommodate a greater number of layers, along with their associated parameters and buffers requiring on-chip memory. On top of that the throughput improvements, as previously discussed, with only a negligible drop in accuracy, further underscore the advantages of this approach. Together, these advancements highlight the effectiveness of this Chapter's approach in achieving higher throughput while maintaining acceptable accuracy levels.

Table 5.5 Comparison of FPGA accelerator architectures generated by SMOF with prior state-of-the-art works on four major computer vision tasks (segmentation, detection, 3D segmentation, action recognition).

	S. Basalama [69]	S. Liu [136]	S. Liu [146]	Vitis-AI [19]	SMOF (Ours)			A. Montgomerie [130]	SMOF (Ours)	P. Toupas [147]	SMOF (Ours)	SMOF (Ours)
Model	UNet†	UNet†	UNet†	UNet†	UNet			YOLOv8s	YOLOv8n	X3D-M	X3D-M	UNet3D
Model Type	2D	2D	2D	2D	2D			2D	2D	3D	3D	3D
Input Size	256 × 256	512 × 512	256 × 256	128 × 128	256 × 256			640 × 640	640 × 640	16 × 256 × 256	16 × 256 × 256	155 × 240 × 240
MACs	12	5.9	27.4	5.6	130			30.5	4.4	6.97	6.97	918.6
Precision	Float32	W16A16	W8A8	W16A16	BFP8			W8A16	BFP8	W16A16	BFP8	BFP8
Board	VCU1525	ZC7045	Arria 10	ZCU102	VCU1525	U200	ZCU102	VCU118	VCU118	ZCU102	ZCU102	U200
Reconfigurations∓	-	-	-	-	-	-	6	-	5	37	9	5
LUT (%)	47.95	38.88	67.54 (ALM)	18.97	83.99	87.96	89.99 (78.67)	86.53	73.28 (46.10)	-	89.71 (86.46)	34.36 (24.61)
DSP (%)	45	71.11	98.69	28.17	87.99	88.62	89.56 (57.62)	99.63	89.09 (74.61)	94.96 (84.43)	48.88 (37.33)	90 (83.17)
BRAM-18K (%)	36.99	50	88.87 (M20K)	27.96	85.99	84.58	89.96 (75.11)	30.6	67.58 (43.43)	94.73 (52.71)	81.68 (52.41)	83.12 (69.54)
URAM (%)	43.95	-	-	-	90	90	-	74.27	57.99 (46.62)	-	-	90 (55)
Frequency (MHz)	234	200	200	281	200	250	200	240	250	160	200	250
Throughput (fps)‡	17.18	17.12	57.47	156.7	16.96	21.21	1.28±	8.33	184.27±	13.44±	27.08±	1.75±
GOP/s	207	107	1578	832	2206	2758	166	1244	808	86	171	1595
GOP/s/DSP	0.06	0.16	0.94	0.22	0.36	0.45	0.11	0.18	0.16	0.03	0.18	0.28

† Optimised UNet architectures tailored to the HW design (variations in MACs)
‡ For 3D CNNs the throughput is defined as *clips/s* instead ± Results on designs with multiple partitions reported with favourable batch size ∓ Designs with '-' indicate no extra reconfiguration of the device after the initial bitstream in loaded

5.6 Conclusions

In this chapter, we addressed the limitations of conventional streaming dataflow architectures for FPGA-based CNN acceleration, particularly in handling large, complex models that exceed on-chip memory available resources. Our proposed memory optimization strategy redefines the role of both on-chip and off-chip memory in layerwise-pipelined, streaming architectures, enabling efficient acceleration of more complex CNNs on FPGAs. The main contributions of this chapter include:

- Efficient Memory Offloading for Streaming Architectures: We introduced a streaming CNN accelerator that supports partial offloading of weights and activations to off-chip memory. This approach mitigates on-chip memory constraints without disrupting the computation pipeline, making it feasible to accelerate larger networks that would otherwise be limited by on-chip resources alone.

- Design Space Exploration (DSE) for Optimised Memory Utilization: We introduced a DSE strategy grounded in a greedy, iterative optimization algorithm that automates the exploration of partitioning configurations and memory usage across on-chip and off-chip resources. This approach ensures optimized resource utilization and performance, particularly for memory-intensive and complex network structures.

- Subgraph-Based Partitioning Methodology: Leveraging FPGA reconfigurability, we developed a subgraph-based partitioning technique that enables effective trade-offs between latency and throughput. Building upon the partitioning strategies introduced in the previous chapter and incorporating the option to offload data to off-chip memory, this method is particularly suited for hierarchical and branched networks, distributing computational load more efficiently and enhancing flexibility in resource allocation.

Our approach demonstrates significant improvements in resource efficiency and scalability by maximizing the use of both on-chip and off-chip memory resources within a streaming dataflow architecture. However, challenges remain when the primary optimization target shifts from throughput to latency. The sequential processing of partitioned designs and the reconfiguration penalties can introduce significant latency overheads, making it challenging for current designs to meet real-time performance requirements.

As a result, while our approach optimizes memory utilization effectively, it remains less competitive in latency-sensitive scenarios, highlighting the need for alternative strategies.

In the following chapter, we address these limitations by investigating an alternative approach which is specifically tailored to the requirements of latency-focused designs. Unlike the throughput-optimized approaches covered in this and previous chapters, the following chapter focusses on a new toolflow designed with latency as its main objective. This unique approach deviates from the partitioning strategies utilised until this point, introducing novel techniques to minimise the latency of the generated designs.

Chapter 6

Navigating New Waters: Latency-Oriented Optimisation for 3D CNNs on FPGAs

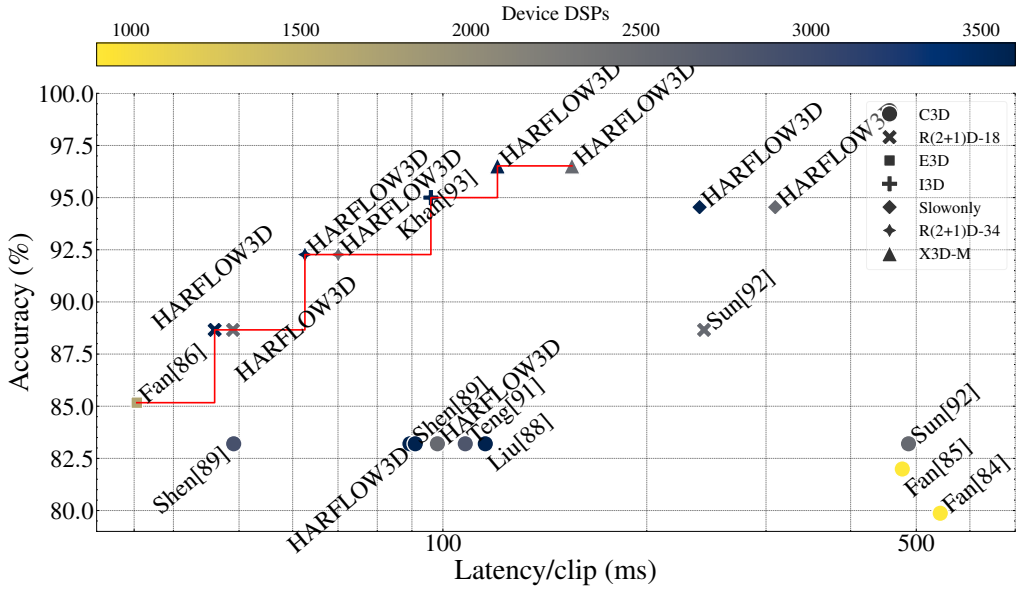
6.1 Introduction

In previous chapters, we demonstrated significant advancements in throughput-oriented optimisation for 3D Convolutional Neural Networks (CNNs) on FPGA devices. Our proposed methodologies effectively maximized resource utilization, balancing on-chip and off-chip memory utilisation to meet high-throughput demands within streaming dataflow architectures. However, throughput-optimized designs often fail to meet the strict timing requirements of latency-sensitive applications. Such applications require real-time performance, while the sequential processing of partitioned designs and the reconfiguration penalties introduce latency overheads that limit system responsiveness.

To address these limitations, this chapter introduces HARFLOW3D[1], a novel toolflow specifically designed to meet the demands of latency-oriented applications. While HARFLOW3D follows the same foundational principles as the toolflow (as shown in Figure 4.2) presented in Chapters 4 and 5—including a model parser, design space exploration (DSE) stage, and hardware mapping component—it introduces two key enhancements tailored for latency optimization. First, a scheduling component has been integrated into the toolflow, as shown in Figure 6.2. Additionally, the hardware building blocks have been updated to incorporate not only compile-time parameters but also new

[1]https://github.com/ptoupas/harflow3d

Fig. 6.1 Pareto front on 3D CNNs: Latency over Accuracy. Designs produced by the proposed HARFLOW3D toolflow dominate the Pareto front.



runtime parameters, enabling them to adapt to multiple layers with varying workloads across the model. This flexible configuration allows HARFLOW3D to support diverse computational demands efficiently, achieving minimized latency for real-time Human Action Recognition (HAR) applications.

Figure 6.1 showcases both prior work and HARFLOW3D designs in terms of achieved accuracy versus latency. HARFLOW3D designs account for most of the points on the Pareto front, demonstrating the toolflow's ability to generate latency-optimized, Pareto-optimal designs across a variety of 3D CNN models for HAR. This Pareto-optimal relationship between accuracy and latency is a valuable asset for design workflows, enabling a designer to finely adjust the balance between model accuracy and latency performance.

The key contributions of this chapter are as follows:

- Introduction of HARFLOW3D, the first 3D CNN-to-FPGA latency-oriented toolflow supporting a wide range of models and devices, achieving latency results competitive with prior hand-tuned implementations.

- Highly-tuned runtime parametrised building blocks are introduced, incorporating the support for feature map tiling and the capability to execute different layers utilising the same underlying hardware in a time-multiplexing manner. These

features enable the proposed designs to achieve significantly lower latency compared to their non-parametrised counterparts.

- Comprehensive evaluation across multiple FPGA devices and 3D CNN models, showcasing up to 5× latency improvement over state of the art FPGA-based approaches.

HARFLOW3D advances FPGA-based acceleration by introducing a latency-optimized toolflow, designed to meet the demands of real-time applications. Through proposed scheduling and runtime parametrisation strategies, we set a new benchmark for efficiently deploying 3D CNNs on FPGAs in latency-sensitive scenarios.

The work presented in this chapter is a collaborative effort, with the contributions distributed as follows. The hardware-level implementation of the runtime-parametrised building blocks and the validation of the corresponding hardware models represent A. Montgomerie-Corcoran's primary contributions. These core hardware contributions are detailed in Sections 6.2.2 and 6.5.1. Figure 6.6 is also A. Montgomerie-Corcoran's contribution. Section 6.3 is a joint contribution from both A. Montgomerie-Corcoran and myself. The remaining aspects of the work, are my own contribution.

6.2 Proposed Architecture

This section outlines the basics of the hardware architecture of the toolflow's generated designs. The proposed toolflow builds upon the streaming architecture principles established in Chapters 3 and 4, while introducing key changes to address latency-oriented demands. The main enhancements include the introduction of runtime-parametrised hardware blocks, enabling each block to process multiple different layers of the 3D CNN model in a time-multiplexing manner. This flexibility allows for optimized latency performance by reducing the need for reconfiguration. Additionally, a new scheduling component has been incorporated as shown in Figure 6.2. This scheduler is responsible for generating the set of scheduling instructions, which are sent from the host CPU to the FPGA. These instructions control the internal flags of the runtime-parametrised hardware blocks, allowing them to adapt to different layers during the execution of the model.

The Neural Network Model Parser, introduced in Section 4.3, is also utilised here as the entry point of the toolflow. The 3D CNN model is converted into an intermediate representation (IR) structured as a DAG, denoted by $M = l_1, ..., l_L$, where l_i represents

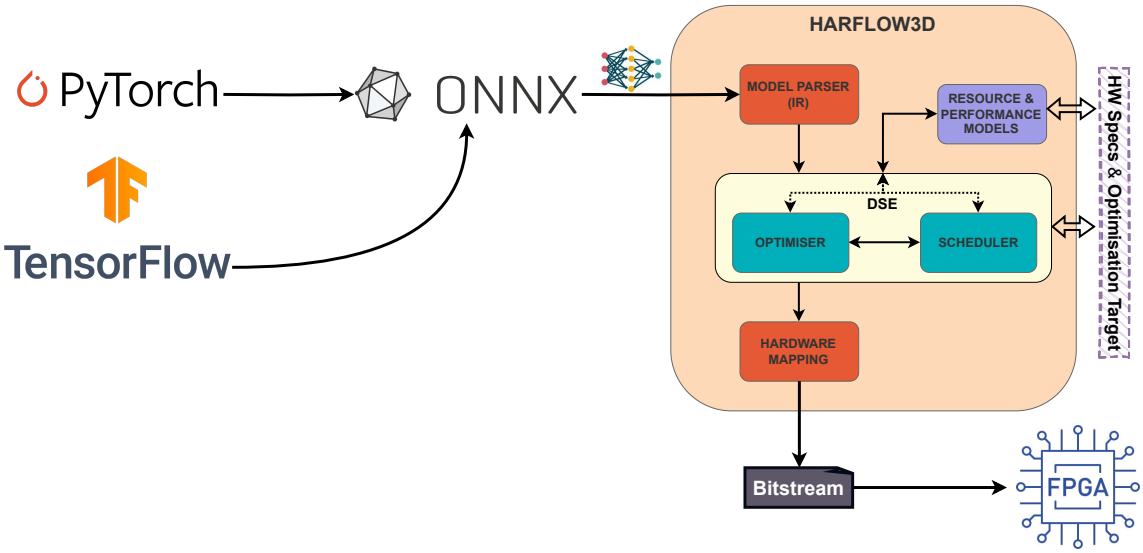


Fig. 6.2 High level overview or the HARFLOW3D toolflow architecture. The four distinct but connected parts are being shown in the diagram. The entry point which is the model parser creates the intermediate representation (IR) and feeds that to the optimiser and the scheduler. During the DSE, the optimiser and scheduler, exchange information with the resource and performance models. Once the optimisation is finished the automated hardware mapping tool converts IR into HDL code and then into a bitstream.

the i^{th} layer within the set of model layers L and is expressed as an logical node of M. The DAG is then translated into a Synchronous Data-Flow Graph (SDFG), denoted as G, with N computation (or hardware) nodes, where $G = n_1, ..., n_N$.

6.2.1 Building Blocks Description

The foundational hardware building blocks used in this toolflow have been introduced and detailed in Section 3.3.1, Section 3.3.2, and Section 4.3.1. In this chapter, we expand on these building blocks by introducing runtime parameters, which provides additional flexibility for adapting computation to varying layer requirements dynamically. Table 6.1 provides a summary of the parameters used for computation and logical nodes to support the descriptions in the following sections.

To further help the reader follow the equations presented in this chapter, we reiterate the information originally introduced in Section 3.3.1 regarding the input and output feature maps. Feature-maps, representing the spatial, temporal, and channel dimensions

of the data processed by each layer, are defined as:

$$\boldsymbol{S_n^{in}} = \{H_n^{in}, W_n^{in}, D_n^{in}, C_n^{in}\}$$
$$\boldsymbol{S_n^{out}} = \{H_n^{out}, W_n^{out}, D_n^{out}, C_n^{out}\}$$

where $H_n^{in/out}$, $W_n^{in/out}$, $D_n^{in/out}$, and $C_n^{in/out}$ represent the height, width, depth, and number of channels for the input and output, respectively.

To clearly distinguish between parameters set at different stages of our toolflow, we adopt a specific notation throughout this chapter. Compile-time parameters, which are fixed during the Design Space Exploration (DSE) to define the physical hardware architecture, are represented by standard symbols (e.g. c_n^{in}). In contrast, runtime parameters, which are dynamically configured during the model's execution to adapt the hardware to a specific layer's workload, are denoted with a circumflex accent (e.g. $\hat{c_n^{in}}$). Furthermore, we use subscripts to differentiate between the hardware and software levels of abstraction. Parameters associated with a hardware hardware node in the hardware graph G are subscripted with n (e.g. c_n^{in}), representing a physical block on the FPGA. Parameters corresponding to a model's logical node (i.e., a specific layer) in the model graph M are subscripted with l (e.g. c_l^{in}). This notation helps to clarify the relationship between the fixed hardware configuration (compile-time, node n) and the dynamic workload it executes (runtime, from layer l). For each type of operation, the set $\hat{\mathcal{V}}$ specifies the runtime parameters associated with its hardware node:

- **Convolution 3D:**

$$\hat{\mathcal{V}}_{Conv} = \{\hat{\boldsymbol{S}}^{in}, \hat{\boldsymbol{S}}^{out}, \hat{\boldsymbol{K}}, \hat{\boldsymbol{J}}, \hat{\boldsymbol{P}}, \hat{Gr}, \hat{c}^{in}, \hat{c}^{out}, \hat{f}\}$$

- **Pooling 3D:**

$$\hat{\mathcal{V}}_{Pool} = \{\hat{\boldsymbol{S}}^{in}, \hat{\boldsymbol{S}}^{out}, \hat{\boldsymbol{K}}, \hat{\boldsymbol{J}}, \hat{\boldsymbol{P}}, \hat{T}, \hat{c}\}$$

- **Global Pooling, Activation & Element-Wise 3D:**

$$\hat{\mathcal{V}}_{GAP} = \hat{\mathcal{V}}_{Act} = \hat{\mathcal{V}}_{EltWise} = \{\hat{\boldsymbol{S}}^{in}, \hat{\boldsymbol{S}}^{out}, \hat{T}, \hat{Q}, \hat{c}\}$$

- **Fully Connected:**

$$\hat{\mathcal{V}}_{FC} = \{\hat{\boldsymbol{S}}^{in}, \hat{\boldsymbol{S}}^{out}, \hat{c}^{in}, \hat{c}^{out}\}$$

6.2.2 Hardware Design and Implementation

The proposed architecture follows the paradigm of a system consisting of a processor extended by a set of custom instructions. The core building blocks described before are equivalent to custom instructions, and their control is performed through a CPU. Each building block is connected to a crossbar that is responsible for handling the routing of data between the building blocks and off-chip memory, as well as performing inter building block routing. The memory access to and from memory is supported through dedicated DMA blocks.

The architecture of the custom instructions follows the Streaming Architecture paradigm, by implementing direct convolution operations and exploring the opportunity of driving the instantiated building blocks as dictated by the 3D model without accessing off-chip memory but at the same time can map multiple operations on the same building blocks in a time-shared manner avoiding the need for reconfiguring the FPGA fabric.

The toolflow is responsible for identifying the required building blocks in order to support the computations of a given 3D CNN model, as well as to tune them in order to optimise the performance of the system given the available resources (FPGA resources and off-chip memory bandwidth). As such, the resulting system is a heterogeneous multi-core system, with blocks tailored to the 3D CNN model and the targeted FPGA device. This deviates from the approach taken in other streaming toolflows such as FINN [23], and fpgaConvNet [24], where the hardware is tailored to specific DNN models with layers being deeply pipelined, utilising bitstream reconfiguration to overcome resource constraints. Such design approach leads to high performance designs for throughput oriented applications, however bitstream reconfiguration inhibits the ability to target latency-driven applications with a latency target smaller than the reconfiguration time of the device, which is usually in the order of hundreds of milliseconds.

Figure 6.3 illustrates a simplified diagram of an example accelerator generated for a given model. Feature-maps are sent to and from off-chip memory via a pair of DMAs, and sent to the hardware nodes through the configurable crossbars. The design has a sandwich-like architecture, with the AXI-Stream crossbars routing data to and from hardware nodes. The output crossbar connects to the input crossbar, allowing for inter-connectivity between hardware nodes.

The fpgaConvNet [24] toolflow is utilised for the implementation of the computation blocks. We regard this as the *baseline* design, where no runtime configuration is supported, and only padded execution can be used to execute variable feature-map sizes. Figure 6.4

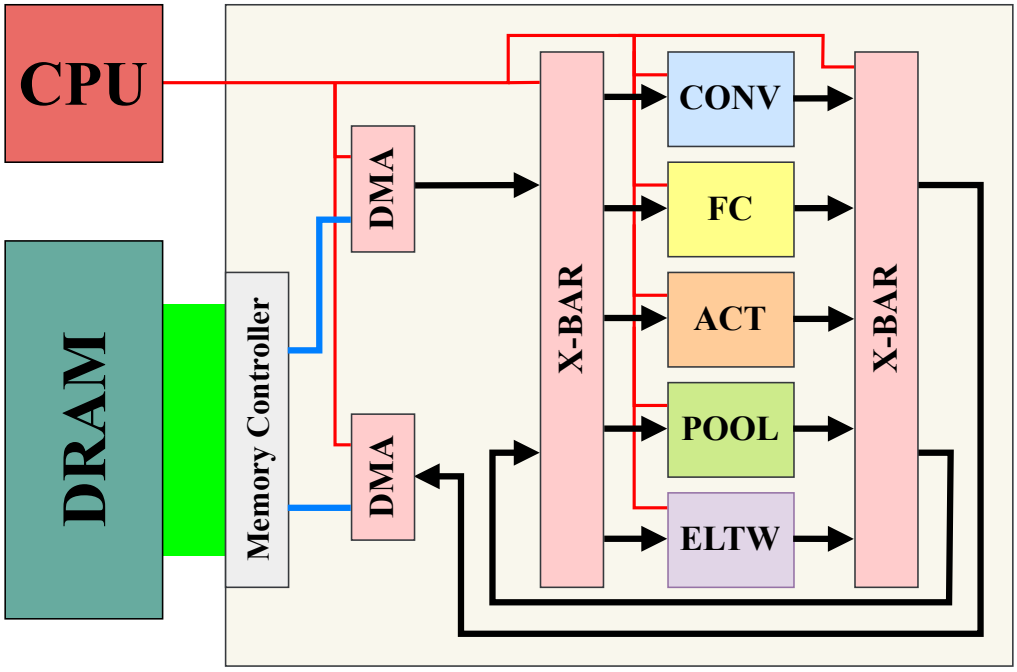


Fig. 6.3 Block diagram of an accelerator instance produced by the HARFLOW3D toolflow. The black lines describe AXI-Stream signals, where the arrows indicate the directionality of the connection, blue are high-throughput AXI interfaces for DMA access, red are AXI-Lite connections for runtime configuration of the hardware nodes, and green indicate the DDR IO interfaces for communicating with off-chip memory.

illustrates how the baseline design has been modified to support runtime configurable layers. The highlighted blocks constitute the overhead required for supporting runtime reconfigurability. The additional hardware includes a configurable counter to change the depth of the line buffers and accumulation buffers, and a crossbar to map configurable kernel sizes to a configurable number of multipliers. The overhead from these additional resources is negligible, and they do not impact the maximum achievable clock frequency.

Unlike the throughput-oriented toolflows of Chapters 4 and 5 which generate HLS C++ code, HARFLOW3D produces hardware described in Chisel[2], an open-source hardware description language (HLD). This decision was made to leverage Chisel's greater flexibility and expressive power, which was crucial for implementing the advanced features required for our latency-oriented designs, such as the runtime-parametrizable building blocks.

[2]https://www.chisel-lang.org/

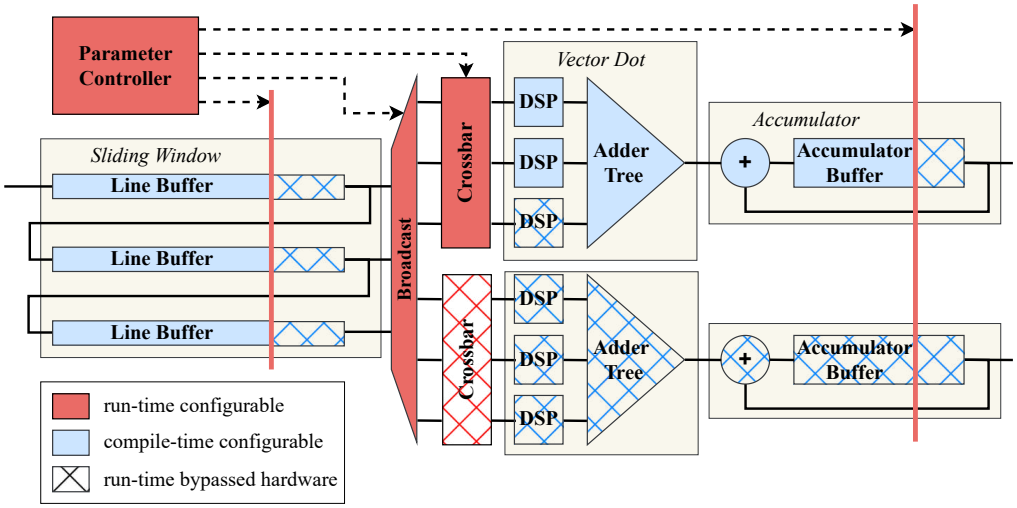


Fig. 6.4 Diagram of hardware for Convolution, and how it can be used with runtime parameters. The blue blocks represent compile-time configurable hardware modules. The red blocks represent runtime configurable hardware modules. Cross-hatching gives an example of how hardware elements can be bypassed at runtime.

Table 6.1 Compile time parameters for node n in the hardware graph G, for each layer type.

	Convolution
F_n	Number of filters (output channel dimension)
$\boldsymbol{K}_n$	3D kernel size (K_n^D, K_n^H, K_n^W)
$\boldsymbol{J}_n$	3D stride (J_n^D, J_n^H, J_n^W,)
$\boldsymbol{P}_n$	3D Padding ($P_n^{D_s}$, $P_n^{D_e}$, $P_n^{H_s}$, $P_n^{H_e}$, $P_n^{W_s}$, $P_n^{W_e}$)
Gr_n	Grouping along the channel dimension
c_n^{in}	parallel streams in
c_n^{out}	parallel streams out
f_n	Vector dot product folding
	Fully Connected
F_n	Number of filters (output channel dimension)
c_n^{in}	Number of parallel streams in
c_n^{out}	Number of parallel streams out
	Pooling
T_n	Type of activation
$\boldsymbol{K}_n$	3D kernel size (K_n^D, K_n^H, K_n^W)
$\boldsymbol{J}_n$	3D stride (J_n^D, J_n^H, J_n^W,)
$\boldsymbol{P}_n$	3D Padding ($P_n^{D_s}$, $P_n^{D_e}$, $P_n^{H_s}$, $P_n^{H_e}$, $P_n^{W_s}$, $P_n^{W_e}$)
c_n	Number of parallel streams in & out
	Activation
T_n	Type of activation
c_n	Number of parallel streams in & out
	Global Average Pooling
c_n	Number of parallel streams in & out
	Element-Wise
T_n	Type of element-wise operation
Q_n	Mode of operation (default or broadcast)
c_n	Number of parallel streams in & out

6.3 Modelling

Precise, high-quality performance and resource modelling is essential to support a rapid and valid design space exploration. In this section, we present a comprehensive, per-layer performance and resource modelling for the runtime parametrised building blocks described in Section 6.2.1. While this modelling follows the foundational principles and concepts outlined in Section 3.4.2, it has been adapted to accommodate the flexibility and dynamic configuration enabled by the newly introduced runtime parameters.

6.3.1 Performance Modelling

Performance modelling is required to evaluate the latency objective during design space exploration. The execution latency of a layer is determined by the dominant bottleneck between its computational workload and the available memory bandwidth. The latency model for each supported hardware type as a function of its runtime parameters $\hat{\mathcal{V}}$ is given as follows:

$$\mathcal{L}_{Conv}(\hat{\mathcal{V}}) = \frac{|\hat{\boldsymbol{S}}^{out}| \cdot \hat{F} \cdot |\hat{\boldsymbol{K}}|}{\hat{c}^{out} \cdot \hat{c}^{in} \cdot \hat{f}}$$

$$\mathcal{L}_{Pool}(\hat{\mathcal{V}}) = \frac{|\hat{\boldsymbol{S}}^{in}| \cdot \hat{C} \cdot |\hat{\boldsymbol{K}}|}{\hat{c}}$$

$$\mathcal{L}_{FC}(\hat{\mathcal{V}}) = \frac{\hat{C} \cdot \hat{F}}{\hat{c}^{in} \cdot \hat{c}^{out}}$$

$$\mathcal{L}_{GAP}(\hat{\mathcal{V}}) = \mathcal{L}_{Act}(\hat{\mathcal{V}}) = \mathcal{L}_{EltWise}(\hat{\mathcal{V}}) = \frac{|\hat{\boldsymbol{S}}^{in}|}{\hat{c}}$$

The preceding models assume unlimited memory bandwidth, however memory accesses have limited throughput. In order to model the effect of memory bandwidth, the consumption and production rates of the layer are required, which are given as,

$$r_n^{in}(\hat{\mathcal{V}}) = \frac{|\hat{\boldsymbol{S}}^{in}|}{\mathcal{L}_n(\hat{\mathcal{V}}) \cdot \hat{c}^{in}}, \; r_n^{out}(\hat{\mathcal{V}}) = \frac{|\hat{\boldsymbol{S}}^{out}|}{\mathcal{L}_n(\hat{\mathcal{V}}) \cdot \hat{c}^{out}}$$

where $r_n^{in}(\hat{\mathcal{V}})$ and $r_n^{out}(\hat{\mathcal{V}})$ are the words per cycle per stream in and out of the hardware node n when executing runtime parameters $\hat{\mathcal{V}}$.

For convolution and fully-connected layers in particular, extra memory bandwidth is required to stream in the weight parameters. This rate is described as,

$$r^{param}_{Conv,FC}(\hat{\mathcal{V}}) = \frac{\hat{C}^{in} \cdot \hat{F} \cdot |\hat{\boldsymbol{K}}|}{\mathcal{L}_{Conv,FC}(\hat{\mathcal{V}}) \cdot \hat{c}^{in} \cdot \hat{c}^{out} \cdot \hat{f}}$$

Alongside the double-buffering of weights, if the channel dimension of a convolution or full-connected layer is folded, the partial sums must be accumulated. This requires streaming the previous partial sum from off-chip memory, whose rate matches the rate out.

$$r^{psum}_{Conv,FC}(\hat{\mathcal{V}}) = r^{out}_{Conv,FC}(\hat{\mathcal{V}})$$

The constrained bandwidth in and out can be described as,

$$BW^{in}_{n}(\hat{\mathcal{V}}) = \min\{BW^{in}_{DMA},\ r^{in}_{n}(\hat{\mathcal{V}}) \cdot \hat{c}^{in}\}$$
$$BW^{out}_{n}(\hat{\mathcal{V}}) = \min\{BW^{out}_{DMA},\ r^{out}_{n}(\hat{\mathcal{V}}) \cdot \hat{c}^{out}\}$$

where $BW^{in}_{n}(\hat{\mathcal{V}})$ and $BW^{out}_{n}(\hat{\mathcal{V}})$ are the constrained words per cycle for the execution of parameters $\hat{\mathcal{V}}$ on the hardware node n. This approach models a performance bound, where the effective on-chip data rate is capped by the available memory bandwidth. The above model summarises the bandwidth in and out for all layers apart from convolution and fully-connected. To describe these, the bandwidth for parameters and partial sums must also be considered,

$$\begin{aligned} BW^{in}_{Conv,FC}(\hat{\mathcal{V}}) = \min\{BW^{in}_{DMA}, & \ r^{in}(\hat{\mathcal{V}}) \cdot c^{in} + \\ & r^{psum}_{Conv,FC}(\hat{\mathcal{V}}) \cdot c^{out} + \\ & r^{param}_{Conv,FC}(\hat{\mathcal{V}}) \cdot c^{in} \cdot c^{out} \cdot f\} \end{aligned}$$

For *Fully Connected* layers, $f = 1$. Given the bandwidths, the total latency for executing a layer is given as,

$$\tilde{\mathcal{L}}_{n}(\hat{\mathcal{V}}) = \max\left\{\frac{|\hat{\boldsymbol{S}}^{in}|}{BW^{in}_{n}(\hat{\mathcal{V}})},\ \frac{|\hat{\boldsymbol{S}}^{out}|}{BW^{out}_{n}(\hat{\mathcal{V}})}\right\} \tag{6.1}$$

It is worth noting that the performance models are functions of runtime parameters, owing to the highly customisable hardware. Streaming Architectures tend to be compu-

tationally bounded, as supported by the results presented in Section 6.5. As the size of the feature-maps is significantly greater than that of the weights, the weight bandwidth is often negligible in comparison to the overall bandwidth required. There is no latency associated with updating the runtime parameters because they are also double-buffered and require negligible information transfer (<100B).

6.3.2 Resource Modelling

Resource modelling is used to explore the performance-resource design space whilst keeping designs within the target FPGA's constraints. Modern FPGA devices share four common resource types: DSP, BRAM, LUT and FF. Only the *Conv* and *FC* layers use DSP resources, An analytical model of DSP usage for building blocks n of these layer types is given by,

$$\mathcal{R}^{DSP}_{Conv} = c^{in}_n \cdot c^{out}_n \cdot f_n \mathcal{R}^{DSP}_{FC} = c^{in}_n \cdot c^{out}_n$$

As a 16-bit fixed-point precision is used throughout the design, each DSP is used for either $16 \cdot 16$ multiplication or multiplication-accumulation. For BRAM modelling, the number of BRAM blocks can be described as,

$$\mathcal{R}^{BRAM}(depth, words) = \left\lceil \frac{depth}{512} \right\rceil \cdot \left\lceil \frac{16 \cdot words}{36} \right\rceil$$

The bus width of the required memory is $16 \cdot words$ as 16-bit fixed-point is used. Only *Conv*, *FC* and *Pool* layers consume BRAM components. Both *Conv* and *Pool* use BRAM for the Sliding Window module, which is described as,

$$\begin{aligned}\mathcal{R}^{BRAM}_{SlW} = \mathcal{R}^{BRAM}\left(W_n \cdot D_n \cdot \frac{C^{in}_n}{c^{in}_n}, (K^H_n - 1) \cdot c^{in}_n\right) + \\ \mathcal{R}^{BRAM}\left(D_n \cdot \frac{C^{in}_n}{c^{in}_n}, K^H_n \cdot (K^W_n - 1) \cdot c^{in}_n\right) + \\ \mathcal{R}^{BRAM}\left(\frac{C^{in}_n}{c^{in}_n}, K^H_n \cdot K^W_n \cdot (K^D_n - 1) \cdot c^{in}_n\right)\end{aligned}$$

Conv and *FC*, require some extra memory for storing weights on-chip, which is modelled as,

$$\mathcal{R}^{BRAM}_{Weight} = \mathcal{R}^{BRAM}\left(\frac{C^{in}_n \cdot F_n \cdot |\boldsymbol{K}_n|}{c^{in}_n \cdot c^{out}_n \cdot f_n}, c^{in}_n \cdot c^{out}_n \cdot f_n\right)$$

where for *Fully Connected* layers, $\boldsymbol{K}_n = \{1, 1, 1\}$ and $f_n = 1$.

For the modelling of LUT and FF resources, a regression model is used due to the non-deterministic nature of FPGA synthesis. The regression models are obtained from a data set of 5000 synthesised modules, where the relationship between the module's parameters and resources are inferred.

Using the derived resource models, we can estimate the resource consumption of a complete hardware graph (G),

$$\boldsymbol{\mathcal{R}}_{total} = \left(\sum_{n \in G} \boldsymbol{\mathcal{R}}_n\right) + \boldsymbol{\mathcal{R}}_{DMA} + \boldsymbol{\mathcal{R}}_{xbar}$$

where $\boldsymbol{\mathcal{R}}$ describes the complete resources (DSP, BRAM, LUT, FF), for the given component. The resource models for DSPs, BRAMs, FFs, and LUTs ($\boldsymbol{\mathcal{R}}$) are the same as those introduced in Chapter 3; however, in this chapter, they are further broken down for a per-layer analysis, offering a more granular view of resource utilization. Additionally, this chapter extends the resource modelling by incorporating two new components not previously included: The DMA and Crossbar (X-BAR) resource models. The former is introduced due to the increased off-chip memory communication demands of the proposed architecture, while the latter captures the resources needed to manage and drive the runtime parameters of the parametrisable hardware building blocks. Both models have been empirically derived from experimental results and are incorporated into the total resource estimation. The quality of the resource model is evaluated in Section 6.5.1.

6.4 Latency-Driven Design Space Exploration

The minimization of the execution latency for a 3D-CNN model on a runtime-configurable accelerator is formulated as an optimization problem. In this context, a comprehensive Design Space Exploration is performed to identify an optimal combination of hardware configurations and a schedule for execution that minimises latency while maximizing resource utilization. Through this exploration, the proposed methodology seeks to balance computational efficiency and adaptability, ensuring that the architecture can respond dynamically to the varied workloads within latency-sensitive applications.

6.4.1 Scheduling

Once an accelerator design has been created, a schedule is needed for executing the 3D-CNN model's layers for this given design. The main choices with regard to scheduling are:

- Mapping of the hardware nodes for executing the 3D-CNN model's layers (i.e. logical nodes).
- Tiling of the feature-maps of logical nodes on a given hardware node.
- Runtime configurations for all the invocations of the hardware nodes based on the respective logical nodes parameters.

The mapping of a hardware node to the set of logical nodes it has to process, is defined by the execution mapping function $\mathcal{E} : G \mapsto \Upsilon(M)$, where $\Upsilon(M)$ is the power-set of M, which is all the distinct subsets of M. This mapping creates disjoint subsets of M, which can be described as,

$$\mathcal{E}(n) \cap \mathcal{E}(m) = \emptyset \quad \forall\, n, m \in G, n \neq m$$

where $\mathcal{E}$ is the execution mapping function. The mapping function must give unique mappings for each hardware node, such that none of the model's layers are executed more than once. The inverse mapping, $\mathcal{E}^{-1}$ finds the corresponding hardware node for a given logical node. The mapping is decided based on the transform described in Section 6.4.3.

Once a mapping has been decided, the schedule for G, denoted as Φ_G, can be created for executing the 3D CNN model graph M on the hardware graph G. This schedule is outlined in Algorithm 4. For each logical node l of the 3D CNN model graph, the tiling factors across each of its dimensions are obtained ($\lceil \frac{H_l^{in}}{H_n^{in}} \rceil, \lceil \frac{W_l^{in}}{W_n^{in}} \rceil, \lceil \frac{D_l^{in}}{D_n^{in}} \rceil, \lceil \frac{C_l^{in}}{C_n^{in}} \rceil$), which are then used to find the tile sizes for computation $(\hat{H}, \hat{C}, \hat{D}, \hat{C})$. The algorithm greedily allocates as much of the feature-map as possible onto the hardware node and then chooses the coarse factors based on the tile shape. Additional runtime parameters such as kernel sizes and padding are also chosen based on the logical node's parameters.

Having created the schedule Φ_G, it can be used to execute the workload. The ordering of dimensions in the proposed accelerator is $NHWDC$, representing the processing order of Batch (N), Height (H), Width (W), Depth (D), and Channel (C), where the channel dimension is the fastest changing, and so the schedule is also executed in this order. The

Algorithm 4 Scheduling Algorithm

$\Phi \leftarrow$ empty list ▷ initialise an empty schedule
for l in M **do**
  $n \leftarrow \mathcal{E}^{-1}(l)$ ▷ get the hardware node
  for $\boldsymbol{i}$ in range($\lceil \frac{H_l^{in}}{H_n^{in}} \rceil, \lceil \frac{W_l^{in}}{W_n^{in}} \rceil, \lceil \frac{D_l^{in}}{D_n^{in}} \rceil, \lceil \frac{C_l^{in}}{C_n^{in}} \rceil$) **do**
    $\hat{H} \leftarrow \min\{H_n^{in}, H_l^{in} - i^H \cdot H_n^{in}\}$
    $\hat{W} \leftarrow \min\{W_n^{in}, W_l^{in} - i^W \cdot W_n^{in}\}$
    $\hat{D} \leftarrow \min\{D_n^{in}, D_l^{in} - i^D \cdot D_n^{in}\}$
    $\hat{C} \leftarrow \min\{C_n^{in}, C_l^{in} - i^C \cdot C_n^{in}\}$
    if $type(n)$ **is** $Conv$ **or** FC **then**
      for $\boldsymbol{i}^F$ in range($\lceil \frac{F_l^{in}}{F_n^{in}} \rceil$) **do**
        $\hat{F} \leftarrow \min\{F_n^{in}, F_l^{in} - i^F \cdot F_n^{in}\}$
        $c^{in} \leftarrow \max\{\ factors\ \hat{C}\ \}$
        $c^{out} \leftarrow \max\{\ factors\ \hat{F}\ \}$
        $\hat{\mathcal{V}} \leftarrow \{\hat{H}, \hat{W}, \hat{D}, \hat{C}, \hat{F}, \hat{c}^{in}, \hat{c}^{out}\}$
        Φ **append** $(n,\ \hat{\mathcal{V}})$
    else
      $c \leftarrow \max\{factors\ \hat{C}\}$
      $\hat{\mathcal{V}} \leftarrow\ \{\hat{H}, \hat{W}, \hat{D}, \hat{C}, \hat{c}\}$
      Φ **append** $(n, \hat{\mathcal{V}})$

total latency for execution is described as,

$$\mathcal{L}_{total}(G) = \sum_{n, \hat{\mathcal{V}} \in \Phi_G} \tilde{\mathcal{L}}_n(\hat{\mathcal{V}}) \tag{6.2}$$

where n is the hardware node and $\hat{\mathcal{V}}$ is the corresponding set of parameters for each configuration in the schedule. The latency $\tilde{\mathcal{L}}_n(\hat{\mathcal{V}})$ of each node n for a configuration $\hat{\mathcal{V}}$ is described in Equation (6.1).

6.4.2 Optimisation Strategy

Simulated annealing (SA), a meta-heuristic for finding minima in non-convex functions, is adopted as the optimisation approach for minimising the latency of 3D CNN model to FPGA mapping. The main objective of the algorithm is to seek the global minimum of a given cost function; in this case, the cost function is latency, which is analytically computed from Equation (6.2). Starting at an initial state by producing random values for the parameters of the hardware nodes, transformations to the hardware graph G are

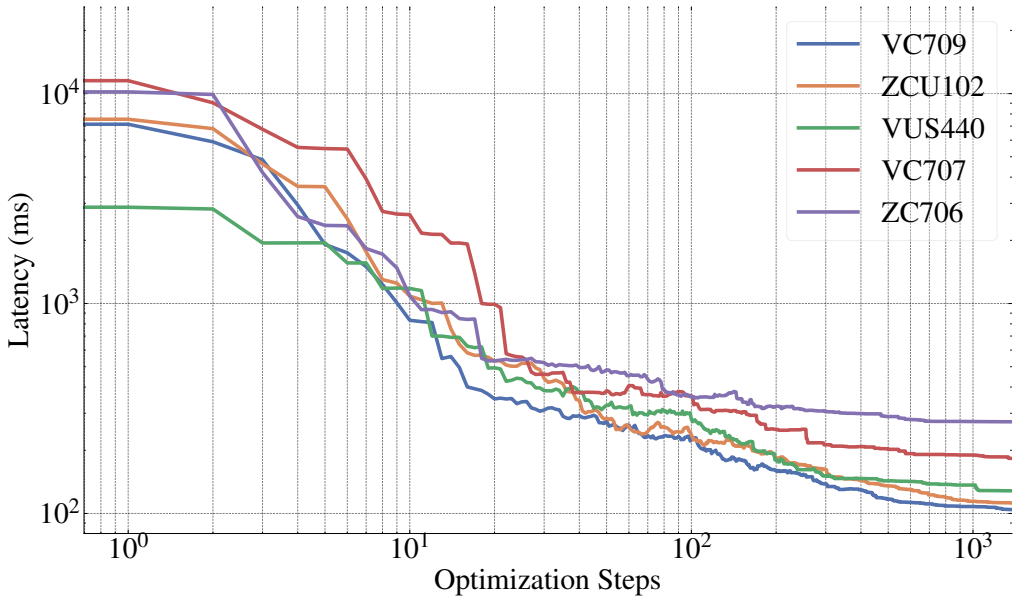


Fig. 6.5 Evolution of latency during Simulated Annealing for various FPGA devices.

applied iteratively generating new states that may be accepted or rejected based on a policy as described in Algorithm 1 in Chapter 3. Each new state is evaluated based on the system's predetermined constraints, and is only accepted if it satisfies all of them. The constraints that must be satisfied on the proposed system are the following:

- The available memory bandwidth should not be exceeded.
- The total used resources $\boldsymbol{\mathcal{R}}_{total}$ should not exceed the available resources of the device.
- The streams in and out (c_n^{in}, c_n^{out}) of each hardware node should be a factor of the channels in and out respectively.
- The parameters for each layer emerged from scheduling must remain within the maximum limits supported by the hardware hardware node responsible for its execution.

Figure 6.5 depicts the evolution of latency during Simulated Annealing. The graph depicts how the latency of the C3D model on various FPGA boards evolves over time. As indicated by the graph, the starting point for each run has extremely high latency, as all of the parameters are set to random values. The latency continues to improve until it reaches a plateau, at which point the optimisation is terminated.

6.4.3 Transformations

In order to traverse the design space, a set of transforms on the hardware graph G are described below.

Feature-Map Dimensions Tiling

To efficiently process neural networks without requiring device reconfiguration, the underlying hardware blocks must accommodate layers with varying feature-map sizes in a time-multiplexing manner. Each layer in the network has unique spatial and channel dimensions, necessitating a common tiling configuration that optimizes the utilization of the hardware block across all layers. By identifying a common factor among the feature-map dimensions of all layers of a certain type of operation, the hardware block can effectively execute operations while avoiding scenarios where smaller feature maps, particularly from deeper layers, lead to underutilization of computational resources. Thus, tiling serves not only to ensure compatibility across layers but also to maximize the utilization of hardware blocks.

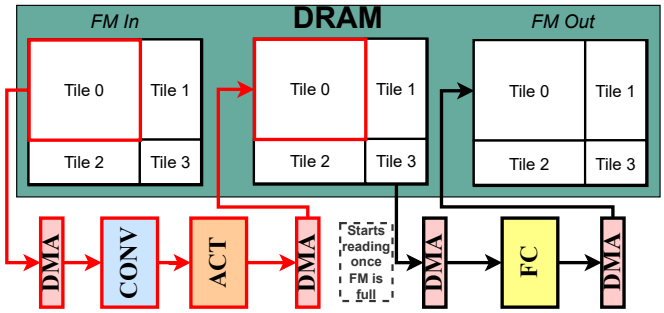
Feature-map dimensions also determine the amount of on-chip memory required for weight parameters and caching, such as the Sliding Window module, as described in Section 6.3.2. Consequently, tiling becomes crucial for addressing the memory limitations of FPGAs. This challenge is further intensified by the presence of multiple layer types, each requiring a dedicated hardware building block, which substantially increases the demands on memory and resource allocation. Directly processing large feature-maps would increase buffering and on-chip memory requirements, making it impractical to efficiently map multiple layers while maintaining high performance. By dividing feature-maps into smaller, manageable tiles that fit within the available memory, hardware nodes can process large feature-maps incrementally, ensuring feasibility and resource efficiency.

Tiling, therefore, serves a dual purpose: it accommodates layer variability and alleviates memory constraints, ensuring high utilization of FPGA resources while enabling the processing of high-dimensional feature-maps.

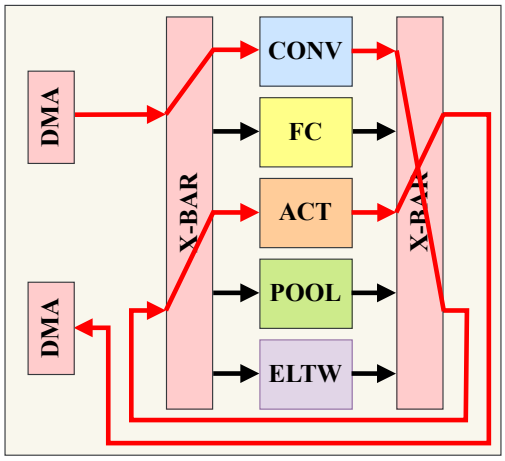
To implement this strategy, a fixed shape for the feature-map dimensions must be assigned to the hardware node at compile time. At runtime, the layer is executed on the hardware node by tiling the feature-map into smaller, manageable segments, as shown in the example in Figure 6.6a. When the optimiser searches for the optimal feature-map shape configuration for the hardware node for both the input and output, the following

conditions must hold,

$$
\begin{aligned}
D_n &\leq max\{D_l : l \in M\} \\
H_n &= max\{H_l : l \in M\} \\
W_n &\leq max\{W_l : l \in M\} \\
C_n &\in \{\ factors\ C_l : l \in M\}
\end{aligned}
$$



(a) 3D CNN Model to SDFG mapping



(b) Hardware execution

Fig. 6.6 The Dataflow of a simple design consisting of Convolution, ReLU, and FC layers. As the red lines and the crossbar dictate the flow between Convolution and ReLU can be addressed within the FPGA without sending the data back to the off-chip memory. This is the result of Fuse Activation optimisation.

For the row dimension (H_n), it is selected as the maximum across all layers because variations in row size do not impact resource usage due to the way the Sliding Window module is implemented. For depth (D_n) and column (W_n) dimensions, any value that is both less than or equal to the maximum across all layers and greater than a minimum

feasible size is considered valid. The channel dimension (C_n) is chosen to be a factor of any of the existing channel dimensions. The tiling parameters, along with other parameters of the hardware node (such as coarse and fine folding, as discussed in Section 6.4.3 below), are explored during the design space exploration (DSE). These parameters are explored by applying the respective transformations to the model graph G, ensuring that the chosen tiling dimensions satisfy the above conditions and minimize the latency of the design.

Coarse-grain and Fine-grain Folding

The concepts of Coarse-grain and Fine-grain Folding, which are key transformations for achieving parallelism in 3D CNN model execution, were thoroughly described in Section 3.4.3. These two folding techniques enhance performance by adjusting the degree of parallelism at both coarse and fine operational levels, achieving a balance between resource utilization and design performance. These folding strategies contribute to the toolflow's ability to explore trade-offs between latency and resource utilization, ensuring designs meet specified performance constraints.

Combination and Separation of hardware nodes

As stated in Section 6.2.2, the toolflow allows several model logical nodes to share the same hardware node. The initial mapping creates unique hardware nodes n for each logical node l. In NNs with multiple layers, this is impractical since the FPGA resources can be quickly exhausted, and the performance of each hardware node would be compromised in order to fit the design. A combination of logical nodes by type for a single hardware node (the available types are depicted in Figure 6.3) is proposed as a solution to this issue. All logical nodes of the same type are combined and mapped onto a single hardware node at the beginning of the optimisation. The compile-time parameters of this node are then modified to ensure it can execute the operations required by its associated logical nodes. This transformation is employed throughout the optimisation procedure, and can affect the computation and logical nodes in two ways:

- Separate hardware nodes: The algorithm chooses L_e logical nodes, where L_e is a hyperparameter, and detaches them from their corresponding hardware node. The new group of logical nodes are integrated and mapped onto new hardware nodes whose characteristics and parameters are adapted respectively.

- Combine hardware nodes: The algorithm searches for hardware nodes of the same type and selects N_c of them, where N_c is a hyperparameter, to combine into a single hardware node. The hardware node's compile-time parameters are updated to support the new set of workloads.

Each time the combination or separation is applied, a set of constraints are considered to assure the result's validity. The required constraints are a combination of the *Feature-Map Dimensions Reshaping, Coarse-grain Folding, and Fine-grain Folding* constraints.

6.5 Performance Evaluation and Results

6.5.1 Hardware Model Validation

Modelling of performance and resources is used extensively for rapid traversal of the large design space. In this section, the accuracy of the performance and resource models is validated across different hardware designs for C3D [3], demonstrating negligible error between modelled and measured results.

Table 6.2 shows a direct comparison between predicted resources and resources after synthesis for a C3D design. The DSP and BRAM models are highly accurate due to the deterministic nature of their synthesis, as resource type annotations are used in the hardware design. For the LUT and FF prediction accuracy, the modelling over-predicts LUT usage and under-predicts FF usage. Logic optimisation contributes to fewer LUTs in the final implemented design, and the additional FF resources likely arise from inter-module buffering that is neglected in the modelling. Additional BRAM resources are

Table 6.2 Comparison of predicted and synthesised resources for C3D designs on a ZCU102 board.

Hardware	**DSP**			**BRAM**			**LUT**			**FF**		
Node	*pred.*	*act.*	*error*	*pred.*	*act.*	*error*	*pred.*	*act.*	*error*	*pred.*	*act.*	*error*
Conv	2304	2304	(+0%)	1052	1052	(+0%)	151K	138K	(+9.4%)	155K	166K	(-6.6%)
MaxPool	0	0	(+0%)	0	0	(+0%)	22K	17K	(+29.4%)	16K	18K	(-11.1%)
Gemm	128	128	(+0%)	456	456	(+0%)	11K	10K	(+1.0%)	15K	18K	(-16.6%)
ReLU	0	0	(+0%)	0	0	(+0%)	1.0K	1.4K	(-28.5)	2.2K	2.2K	(+0%)
DMA		0			51			2.9K			4.7K	
X-BAR		0			0			1.7K			1.4K	
Total	2432	2432	(+0%)	1559	1559	(+0%)	189K	171K	(+7.8%)	194K	210K	(-9.4%)
Avail.		*(2520)*			*(1824)*			*(274K)*			*(548K)*	

required by DMAs for the buffering of bursts across the feature-map. This is accounted for during optimisation.

The convolution layers dominate the total resource consumption, with DSPs typically being the limiting factor. The convolution layers are investigated further, where statistical resource modelling information is captured over 16 designs with varying configurations among different layers. Supporting the results of Table 6.2, the Mean Percentage Error (MPE) and the Standard Deviation (σ) over the 16 different convolution configurations are shown in Table 6.3.

Table 6.3 Statistical resource modelling information over multiple synthesis runs for different convolution layers and configurations.

	DSP	BRAM	LUT	FF
MAPE (%)	0.0	0.35	7.21	8.81
σ	0.0	0.38	8.82	2.89

The accuracy of the performance model is evaluated in Figure 6.7, where the calculation of the percentage error is given by: $error = \frac{|Predicted - Measured|}{Measured} * 100$. The small percentage differences between predicted and measured results on ZCU106 board imply a good level of accuracy and confidence in the toolflow's modelling results. The divergence between the expected and actual latency of the layers presented in Figure 6.7 is due to the DMA introducing a delay between bursts due to memory access cycles. The MAPE for all convolution layers of the C3D model at the particular design is 6.64%.

6.5.2 Performance Evaluation

This section evaluates the proposed methodology's effectiveness in identifying optimal latency-oriented designs. The automatically generated designs are benchmarked against existing 3D CNN accelerators, using the same diverse set of 3D CNN models presented in Chapter 4, specifically detailed in Table 4.2. Each model in this evaluation has shown state-of-the-art performance across various HAR benchmarks, encompassing a range of workload sizes and network complexities. Further details are available in Section 4.4.1 of Chapter 4.

Ablation Study

In this section, an ablation study was undertaken to assess the effect of several optimization strategies on the final performance of the proposed method. Having a baseline strategy

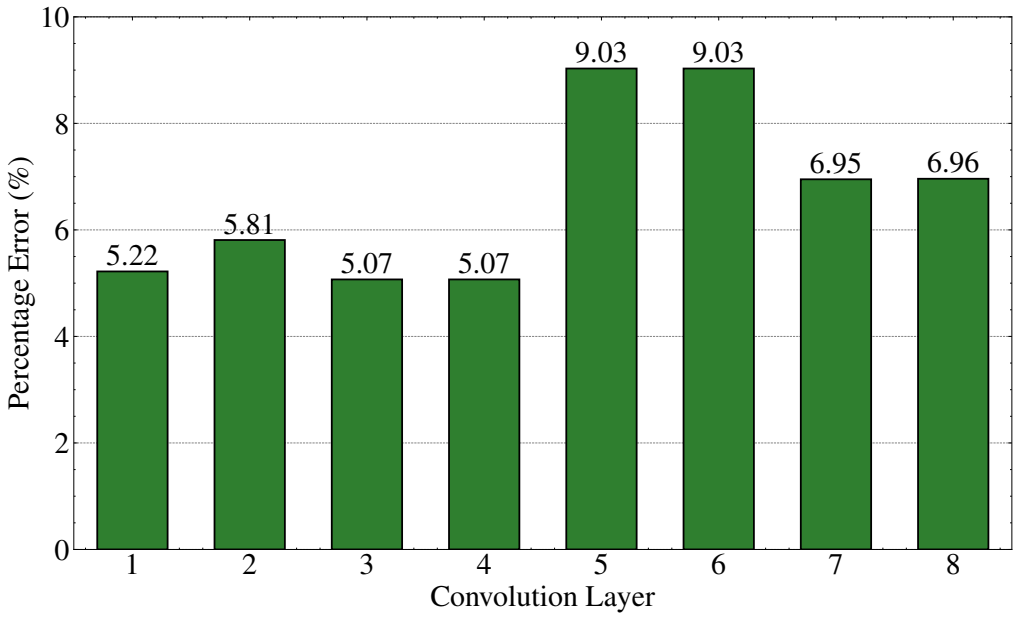


Fig. 6.7 Comparison of predicted latency to measured latency for all the convolution layers of C3D on the ZCU106 board, depicted as percentage error.

in place, as well as introducing and investigating modifications and additions to it, has yielded significant insights and conclusions regarding the direction that should be followed for improving the optimization process. For the ablation experiments, the R(2+1)D-18 model was used, although the findings apply to all supported models and devices.

Baseline Design Description:

The baseline optimisation strategy is defined as follows: the Simulated Annealing (SA) hyperparameters are configured by, $\tau_{start} = 10$, $\tau_{min} = 1 \times 10^{-6}$, and the cooling rate $\kappa = 0.99$, while a warm start is executed prior to the execution of the optimiser. These parameters remain constant throughout the ablation study. For the baseline experiments, the *Feature-Map Dimensions Reshaping, Coarse-grain Folding, and Fine-grain Folding* transformations are enabled while the use of runtime parameters and the fusion of activation layers to preceding layer are disabled.

Optimisation Strategies:

- Building Blocks Combination: Incorporating the *Combination and Separation of hardware nodes* transformation into the optimisation approach improved the optimiser's performance by 1.14×. This optimisation is described in detail in Section 6.4.3.

- Fusion of activation functions into previous layer: Upon revisiting the optimisation findings and assessing each layer type of the model, it was found that the activation layers are typically memory bound, which hinders the overall performance of the design. To overcome this constraint, the fusion optimization was introduced. Through the crossbars seen in Figure 6.6b, such layers are fused directly to the preceding ones (mostly convolution layers). Since convolution layers are mainly compute bound, the fused activation layers also become compute bound, suppressing the memory bound limitation. The addition of this specific optimisation has provided a 1.52× boost in performance.

- Runtime reconfiguration of layer parameters: The introduction of runtime reconfigurability of hardware nodes resulted in the most significant improvement boost. As described in Section 6.2.2, in a non-runtime parametrisable node, in order to support a layer with different runtime feature-map dimensions the underlying hardware modules would need to add padding to match the compile-time dimensions. This affects the performance of the hardware node greatly, as it must perform redundant operations. With the introduction of runtime parametrisable modules, as shown in Figure 6.4, the extra cost of padding and all the redundant operations are omitted, resulting in a 18.21× performance increase.

Resources against Latency Comparison

Figure 6.8 depicts the Pareto front between resources and latency. The DSP utilisation was chosen to represent resources, as the generated designs are typically limited by the number of available DSPs. The figure shows the optimiser's ability to traverse the resource-latency design space, achieving fine-grain control over this trade-off. Indeed the optimiser is able to double performance along the Pareto front at the cost of double the number of DSPs, exploring many design points in between.

Comparison to the state-of-the-art

Having evaluated the performance of the hardware and the effectiveness of the design space exploration method, the proposed work is positioned against existing accelerator works. Table 6.4 outlines the current space for FPGA-based 3D CNN acceleration, and how HARFLOW3D compares. Please note that all existing accelerators are the product of approaches that target only the specific workload, and thus are hand-tuned, where the produced designs from HARDFLOW3D are generated by a single toolflow. Leveraging

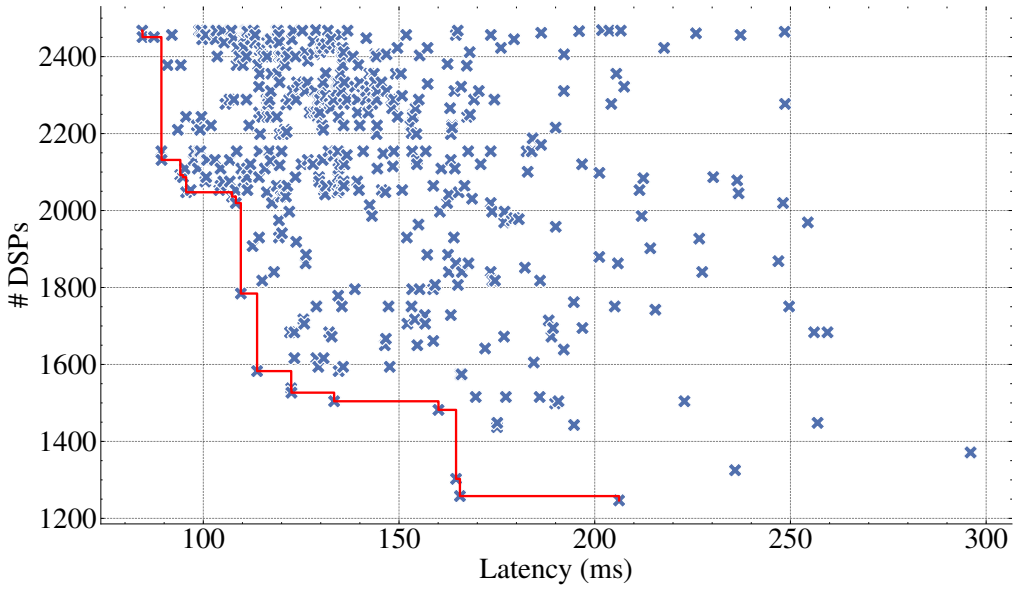


Fig. 6.8 Pareto front of DSP utilisation against latency for the R(2+1)D-34 model on a ZCU102 platform during Simulated Annealing optimisation.

the parametrisation of the hardware blocks and the design space exploration (DSE) process, we achieved outstanding performance across a diverse set of networks, surpassing the performance of existing accelerators, as shown in Table 6.4.

Figure 6.9 provides a more extensive and direct comparison to existing works on C3D model. Since HARFLOW3D can target all of the platforms that previous studies have targeted for C3D, results for each of them have been collected for a direct comparison with all of the existing works. The results are evaluated in DSP efficiency, GOPs normalised over the device's DSPs. As shown in Figure 6.9, HARFLOW3D achieves 1.89× increased DSP efficiency on ZC706 compared to H. Fan [110]. On ZCU102, 5.03× greater results are achieved than M. Sun [116]. On VC709, HARFLOW3D achieves 1.27× better DSP efficiency than Z. Liu [113]. Compared to T. Teng [115] on VC707, the DSP efficiency is 1.48× lesser, however the comparison cannot be considered direct since the specific design uses 8-bit fixed-point arithmetic precision. A direct comparison with J. Shen [2] was not feasible since the authors used a different and smaller version of C3D compared to the original one [3] used in the rest of the works, hence it is not included in this discussion.

The latency improvements achieved in this chapter are significant across all evaluated models compared to those from Chapter 4. The results from Chapter 4 correspond to designs generated by the toolflow when targeting a batch size of 1, without any specific instructions to minimize latency (e.g. trying to minimise the number of partitions). For the C3D model, latency decreases from 787.69 to 98.15 ms. Similarly, the SlowOnly

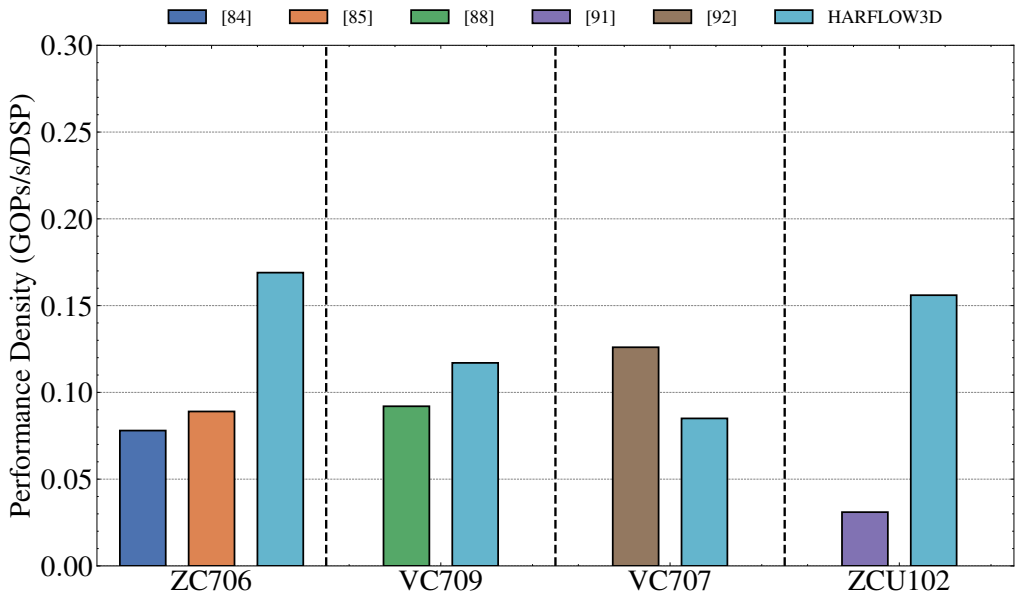


Fig. 6.9 Comparison of the DSP efficiency between HARFLOW3D and previous works. The wide range of the supported FPGA platforms allows for a direct comparison. We demonstrate competitive, and in some cases superior performance.

model's latency drops from 2845.97 to 309.56 ms, while R(2+1)D-18 sees a reduction from 1851.48 to 48.99 ms. Finally, the X3D model shows a latency improvement from 3101.73 to 155.07 ms. Overall, the geometric mean improvement in latency across these four models, comparing the designs from this chapter to those in Chapter 4, is 15.36×, underscoring the effectiveness of the runtime reconfiguration approach in achieving low-latency performance for 3D CNN models. This substantial improvement is primarily due to the runtime reconfiguration blocks introduced in this chapter, which eliminate the need for full device reconfiguration and the associated latency penalties. By enabling on-the-fly adjustments, these blocks provide the flexibility needed to adapt efficiently to the diverse layer requirements of the 3D CNN models, resulting in the significant latency reductions observed.

Table 6.5 outlines a comparison between an Nvidia RTX 2060, a mid-range GPU with 1920 CUDA cores, and a ZCU106, a mid-range FPGA evaluation board featuring the Zynq UltraScale+ XCZU7EV device. The energy consumption per clip for the GPU is $2.02J$, while our design achieves $1.72J$, reflecting a 1.17× improvement in energy efficiency. This underscores the potential and advantages of our proposed approach in addressing the challenges of 3D CNN computational workloads.

Table 6.4 Comparison of HARFLOW3D with existing works, for 3D CNN HAR models. Our tool can target all models, including X3D.

Architecture	H. Fan [109] Hand-Tuned	H. Fan [110] Hand-Tuned	Z. Liu [113] Partial*	T. Teng [115] Hand-Tuned	J. Shen [2]† Partial*		M. Sun [116] Partial*		H. Fan [111] Hand-Tuned	F. H. Khan [117] Hand-Tuned	HARFLOW3D (Ours) Toolflow									
Model	C3D	C3D	C3D	C3D	C3D		C3D	R(2+1)D-18	E3D	I3D	C3D		Slowonly		R(2+1)D-18		R(2+1)D-34		X3D-M	
GMACs	38.61	38.61	38.61	38.61	-		38.61	8.52	6.1	110	38.61		54.81		8.52		12.91		6.97	
Precision	W16A16	BFP24	W16A16	W8A8	W16A16	W16A16	W16A16	W16A16	Float32	W8A8	W16A16	W16A16	W16A16	W16A16	W16A16	W16A16	W16A16	W16A16	W16A16	W16A16
Accuracy (%)	79.87	81.99	83.2	83.2	83.2		83.2	88.66	85.17	95	83.2		94.54		88.66		92.27		96.52	
Board	ZC706	ZC706	VC709	VC707	VC709	VUS440	ZCU102	ZCU102	Intel SX660	VC709	ZCU102	VC709	ZCU102	VC709	ZCU102	VC709	ZCU102	VC709	ZCU102	VC709
DSP (%)	90	86.6	99.8	96	42	53	48	48	93.3	100	96.51	97.77	62.38	69.11	96.94	97.91	95.67	97.25	53.52	89.61
BRAM-18K (%)	86.6	88.1	26.6	25.3	52	30	100	100	-	79	71.93	63.33	78.56	81.05	69.13	64.62	69.24	54.52	56.14	78.67
Frequency (MHz)	172	200	120	160	150	200	150	150	150	200	200	150	200	150	200	150	200	150	200	150
Latency/clip (ms)	542.5	476.8	115.5	107.9	89.4	49.1	487	243	35.32	96	98.15	91.03	309.56	239.34	48.99	46.02	70.05	62.55	155.07	120.38
GOps/s	71.17	80.97	334.28	357.83	431.87	786.35	79.28	35.06	172.8	1145.83	393.37	424.14	177.05	229.01	173.91	185.13	184.29	206.39	43.78	56.14
GOps/s/DSP	0.079	0.089	0.092	0.127	0.119	0.273	0.031	0.013	0.102	0.318	0.156	0.117	0.07	0.063	0.069	0.051	0.073	0.057	0.017	0.015
Op/DSP/cycle	0.459	0.449	0.773	0.798	0.799	1.365	0.209	0.092	0.68	1.59	0.781	0.785	0.351	0.424	0.345	0.342	0.365	0.382	0.086	0.104

* Proposed design supports multiple models (both 2D and 3D), although being tailored to the characteristics of specific 3D CNN models. † The C3D model used is different/smaller version from the original one [3].

Table 6.5 Comparison of HARFLOW3D against Nvidia RTX 2060 GPU on the C3D model.

	GPU	FPGA (HARFLOW3D)
Platform	RTX 2060	ZCU106
Clock Frequency	1.2 GHz	200 MHz
Memory Capacity†	6 GB	4 GB (38 Mb)
Memory Bandwidth	336 GB/s	19.2 GB/s
Precision	Float32	W16A16
Latency/clip (ms)	25.96	182.81
Power (W)	77.97	9.44
Energy/clip (J)	2.02	1.72

† For the ZCU106 the number in the parenthesis indicates the on-chip memory capacity.

6.6 Conclusions

In this chapter we introduced HARFLOW3D, a latency-oriented toolflow for mapping 3D Convolutional Neural Networks (CNNs) onto FPGAs. Targeting real-time applications, HARFLOW3D addresses the specific challenges of minimizing latency in FPGA-based acceleration for Human Action Recognition (HAR) tasks. Unlike the throughput-oriented methods presented in the previous chapters, this chapter focused on strategies to reduce latency by leveraging runtime configurability and enhanced scheduling methodologies. Following are the contributions of this chapter:

- Development of HARFLOW3D, the first latency-oriented 3D CNN to FPGA toolflow, capable of supporting a wide variety of models and devices. HARFLOW3D integrates both compile-time and runtime parameterisation, significantly improving the adaptability and latency performance compared to previous hand-tuned designs.
- Introduction of highly-tuned, runtime-parametrized building blocks that adapt on-the-fly to varying layer requirements within a 3D CNN model, reducing device reconfiguration overheads.
- Comprehensive evaluation across multiple FPGA devices and 3D CNN models, with HARFLOW3D demonstrating up to 5× improvement in latency performance over existing FPGA-based approaches. In particular, the average latency improvement across all evaluated models compared to the Chapter 4 designs is 15.36×, underscoring the impact of the runtime reconfiguration and scheduling techniques.

These advancements position HARFLOW3D as a significant step forward in enabling FPGA-based acceleration for latency-sensitive applications. By setting a new benchmark for low-latency FPGA deployments, HARFLOW3D enhances the feasibility of deploying complex 3D CNN models in real-time settings.

One notable limitation in the current HARFLOW3D approach is the per-layer execution model, which relies on interactions with the host CPU and off-chip memory. This dependence can create bottlenecks, as the frequent off-chip communication may add delays and stalls to the execution of the model. To address this, a potential enhancement could involve developing larger processing blocks that encapsulate multiple layers of the model. Such a design would reduce the need for frequent off-chip memory accesses by keeping data on-chip across multiple layers, thus reducing delays and increasing device resources efficiency. At the same time, these larger blocks could still support

runtime configurability to maintain flexibility across different parts of the 3D CNN model. This hybrid approach could combine the advantages of the deeply pipelined architectures of throughput-oriented designs (as explored in previous chapters) with the runtime configurability of the hardware blocks as presented in this chapter, potentially delivering performance improvements when targeting both high throughput and low latency applications.

Chapter 7

Conclusions

As we reach our final destination, it is time to reflect on the path traversed and the lessons learned throughout this journey, which have shaped the key contributions and findings of this thesis, highlighting advances made in the field of FPGA-based acceleration for 3D Convolutional Neural Networks (CNNs). Through the development of innovative methodologies, toolflows, and optimization strategies, this work has addressed critical challenges in efficiently mapping and accelerating modern 3D CNNs models. This chapter summarises the key contributions of this thesis, linking the findings of the previous chapters to the broader context of AI hardware acceleration. It also outlines potential directions for future research, expanding on the foundations established in this work to address emerging challenges in this rapidly evolving field. Finally, the chapter concludes with reflections on the significance of this work and its place within the larger context of AI innovation.

7.1 Summary of Contributions

A Primer on FPGA-Based 3D CNN Model Optimisation

Chapter 3 explored the mapping and optimization of 3D Convolutional Neural Networks (CNNs) on FPGAs, focusing on the challenges arising from their high-dimensional data, which imposes higher computational and memory demands compared to that of 2D Convolutional Neural Networks. This chapter constitutes a case study on the X3D model, a state-of-the-art 3D CNN for Human Action Recognition (HAR) at the time of writing this thesis, chosen for its complex architecture and its state-of-the-art performance. The choice of this model was motivated by the limitations of existing FPGA-based solutions

in handling such complex architectures and the lack of approaches specifically targeting 3D CNNs.

To tackle these challenges, this chapter presented a novel streaming-based hardware architecture for X3D, extending the Synchronous Data Flow (SDF) graph representation previously used for 2D CNNs [24] to incorporate and support 3D CNN building blocks, along with performance and resource models for these components. Leveraging a Design Space Exploration (DSE) approach, a design was proposed that aligns with the specific characteristics of the X3D model and the constraints of the target FPGA platform. By showcasing the efficient deployment of X3D, the methodologies introduced in this chapter established a foundation for the development of the subsequent toolflows.

Although initially designed for X3D, the methodologies presented in this chapter, serve as a basis for accommodating a wider range of 3D CNN model architectures, showcasing their adaptability and versatility. This chapter set the stage for the toolflows and techniques developed in subsequent chapters, highlighting the potential of reconfigurable hardware to meet the demands of modern AI workloads.

Streamlining FPGA-Based Optimisation for 3D CNNs Targeting Throughput-Oriented Designs

Building on the foundations established in Chapter 3, Chapter 4 introduced fpgaHART, a throughput-oriented toolflow designed to streamline and automate the mapping and optimization of 3D CNNs on FPGAs. The optimization process, particularly the manual partitioning of models, can be time-consuming, and may not always lead to optimal designs. The motivation behind the introduction of this toolflow was twofold: to model the partitioning process as an optimization problem, thereby reducing the manual effort, and to provide a fast and automated approach for mapping and optimizing various 3D CNN models across a range of FPGA devices.

The key innovations of this chapter include an automated model partitioning strategy that iteratively identifies and refines optimal reconfiguration points, incorporating guidance from user-provided hints. This strategy also facilitated the Design Space Exploration (DSE) methodology introduced in Chapter 3, expanding its capabilities to manage multi-branch and multi-input, multi-output partitions, made possible by the flexibility to set reconfiguration points at arbitrary positions within the network. Furthermore, the set of supported operations introduced in Chapter 3 was extended to accommodate the 3D CNNs evaluated in this chapter, including C3D, SlowOnly, R(2+1)D-18, R(2+1)D-34. These contributions enabled fpgaHART to deliver significant throughput improvements

across various FPGA platforms and 3D CNN models, establishing a solid foundation for high-throughput AI accelerators tailored to 3D CNNs.

Chapter 4 showcased the effectiveness of automation in optimizing complex 3D CNN architectures to achieve high throughput designs. By expressing the partitioning process as an optimization problem, the proposed approach reduced the manual effort and enabled the development of scalable and efficient design solutions. This chapter established a solid foundation for automated partitioning and optimization, marking a significant step forward in FPGA-based acceleration for high-performance AI workloads.

Enhancing CNN Performance by Leveraging Off-Chip Memory Utilisation on FPGAs

Chapter 5 extended the capabilities of the fpgaHART toolflow, enabling it to explore a broader design space. This chapter addresses the challenge of limited on-chip memory in FPGA-based streaming architectures, a constraint that often hinders the handling of large and complex 3D CNN models with deep skip connections and significant buffering requirements.

To address these challenges, this chapter introduced a memory optimization strategy that revisits the utilization of both on-chip and off-chip memory in layerwise pipelined streaming architectures. Key innovations include integrating additional memory-related options into the DSE process, such as activation eviction to off-chip memory and weight fragmentation for convolutional layers, enabling partial offloading of layer weights to off-chip memory. Additionally, an expanded subgraph-based partitioning approach was developed, incorporating the option to offload data to off-chip memory. These additions allowed for improved memory efficiency while reducing computation pipeline stalls.

Chapter 5 addressed the critical challenge of limited on-chip memory in FPGA-based streaming architectures. By leveraging both on-chip and off-chip memory resources, this chapter demonstrated how modern 3D CNN models can be efficiently handled without compromising performance. The advancements made here not only maximize the memory efficiency of the produced designs but also provide a scalable solution for tackling increasingly demanding AI workloads on FPGAs.

Latency-Oriented Optimisation for 3D CNNs on FPGAs

In contrast to the previous three chapters, Chapter 6 focused on the challenge of optimizing 3D CNNs for latency-sensitive applications, such as real-time human action

recognition. Throughput-oriented designs often introduce latency overheads due to sequential processing and reconfiguration penalties, necessitating a different approach tailored to minimizing latency. To address these limitations, this chapter introduced HARFLOW3D, a latency-oriented toolflow specifically designed to meet the demands of latency-critical applications.

The key innovations of this chapter include the introduction of highly-tuned, runtime-parametrizable hardware blocks. These blocks incorporate feature map tiling and support the execution of multiple layers using the same underlying hardware. Runtime parameterisation is achieved through custom instructions sent from a host machine, derived from a static schedule generated during model optimization. This approach eliminates the need for device reconfiguration, enabling HARFLOW3D to achieve significant latency reductions while delivering competitive, and even state-of-the-art, performance compared to other existing works in the field. The toolflow was evaluated across multiple FPGA platforms, achieving state-of-the-art performance and demonstrating substantial improvements over existing approaches.

In summary, Chapter 6 tackled the critical challenge of optimizing 3D CNNs for latency-sensitive applications, providing a novel solution with the HARFLOW3D toolflow. By introducing runtime-parametrizable hardware blocks and leveraging static scheduling to eliminate reconfiguration overheads, this chapter demonstrated the feasibility of achieving significant latency reductions without compromising performance. HARFLOW3D demonstrated state-of-the-art performance, setting a new benchmark for FPGA-based acceleration in latency-critical, real-time applications.

7.2 Future work

This thesis has focused on leveraging the reconfigurable nature of FPGAs to develop optimised and efficient solutions for executing the workloads of 3D Convolutional Neural Networks (CNNs), addressing their computational and memory challenges in both throughput- and latency-oriented designs. While significant progress has been made, there are still promising opportunities for future research to expand upon the foundations established in this study.

Exploring Hybrid Architectures

A natural extension of this research lies in the development of hybrid architectures that combine the strengths of throughput- and latency-oriented designs. This could be

accomplished by enhancing the parametrized hardware blocks introduced in Chapter 6 to incorporate multiple layers within each block, internally pipelined in a way similar to the designs presented in Chapters 4 and 5. This hybrid approach would further expand the design space, allowing the exploration of novel configurations that neither the latency-focused nor throughput-focused toolflows could previously consider. This enhancement would open new possibilities for fine-tuning the balance between latency and throughput while maintaining high utilization of FPGA resources, paving the way for more versatile FPGA-based solutions.

Enhancing Design Space Exploration Algorithms

Another promising direction involves improving the Design Space Exploration (DSE) optimisation algorithms. Although combining such algorithms with heuristics has proven effective in generating high-performance designs, these strategies can be further enhanced to more systematically and efficiently navigate the vast design space. Specific improvements could include:

- **Advanced Search Strategies**: Future work could explore the integration of more sophisticated, learning-based optimisation techniques. For instance, Reinforcement Learning (RL) could be employed to train an agent that learns an optimal policy for applying transformations to the SDFG. Such an agent could potentially discover non-intuitive, high-performance configurations that heuristic methods might miss, while also reducing the time required to find a solution.
- **High-Fidelity Cost Models**: The accuracy of the DSE is fundamentally tied to the precision of its performance and resource models. A valuable research direction would be to develop more accurate cost models using machine learning. By training a model on a large dataset of post-synthesis results from various hardware configurations, it would be possible to create predictors that more accurately estimate resource utilisation (LUTs, FFs) and performance metrics, leading to better-informed decisions during the DSE process.

Adapting to Vision Transformers

Finally, while this thesis targeted 3D CNNs, the growing significance and potential of Vision Transformers (ViTs) cannot be overlooked. ViTs mark a significant architectural shift, relying on the self-attention mechanism, which introduces unique hardware acceler-

ation challenges. The methodologies developed in this thesis provide a strong foundation for addressing these challenges:

- **Hardware Acceleration of the Self-Attention Mechanism**: The primary computational bottleneck in ViTs is the self-attention module, which has a quadratic complexity with respect to the input sequence length. The streaming architecture approach in this thesis is well-suited to tackle this. A key research avenue would be to design a dedicated, highly-pipelined hardware building block for the self-attention mechanism, similar to how convolutional layers were handled. This block would be optimised for the large matrix multiplications (Query, Key, and Value projections) and the subsequent softmax and value aggregation steps.

- Managing Memory for Large Tensors: The self-attention mechanism generates large intermediate tensors, such as the attention score matrix, which can easily exceed on-chip memory capacity. The memory management strategies developed in Chapter 5, specifically activation eviction and weight fragmentation, could be directly adapted to this challenge. These techniques would allow the system to intelligently offload large attention maps to off-chip memory, mitigating on-chip memory bottlenecks without stalling the computational pipeline, which is a critical issue for ViT acceleration.

Through these extensions, the contributions of this thesis could be expanded to tackle an even broader array of challenges, ensuring the continued relevance and applicability of FPGA-based solutions in the evolving field of AI hardware acceleration.

7.3 Final Thoughts and Reflections

The journey undertaken in this thesis represents a step toward in addressing the ever-growing computational and memory demands of modern AI models. By leveraging the flexibility of FPGAs and focusing on model-agnostic solutions, this work has demonstrated how the reconfigurable nature of these devices can be utilised to meet the challenges posed by state-of-the-art machine learning models deployed in resource-constrained environments.

The remarkable growth of AI has brought unprecedented capabilities while at the same time has created significant demands for computational efficiency. By addressing these challenges, this thesis contributes to a larger body of work that seeks to make AI systems more accessible, sustainable, and efficient in real-world settings. While the

focus of this thesis has been on 3D CNNs, with the specific contributions tailored to these workloads, the methodologies and approaches developed offer a basis that can be extended to other domains, addressing the diverse and emerging challenges in AI acceleration. As AI continues to evolve, this work underscores the importance of curiosity and exploration in driving research forward.

References

[1] K. Soomro, A. R. Zamir, and M. Shah, "UCF101: A Dataset of 101 Human Actions Classes From Videos in The Wild," 12 2012.

[2] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, "Towards a uniform template-based architecture for accelerating 2d and 3D CNNs on FPGA," in *FPGA 2018 - Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, vol. 2018-Febru, pp. 97–106, 2018.

[3] S. Ji, W. Xu, M. Yang, and K. Yu, "3D Convolutional neural networks for human action recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 1, pp. 221–231, 2013.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, IEEE, 6 2009.

[5] G. J. Brostow, J. Fauqueur, and R. Cipolla, "Semantic object classes in video: A high-definition ground truth database," *Pattern Recognition Letters*, vol. 30, pp. 88–97, 1 2009.

[6] C. Feichtenhofer, "X3D: Expanding Architectures for Efficient Video Recognition," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (Feichtenhofer Christoph, ed.), pp. 200–210, 2020.

[7] T. Sveinsdottir, P. Troullinou, S. Aidlinis, A. Delipalta, R. Finn, P. Loukinas, J. Muraszkiewicz, R. O'Connor, K. Petersen, M. Rovatsos, N. Santiago, D. Sisu, M. Taylor, and P. Wieltschnig, "The Role of Data in AI," 12 2020.

[8] A. Aldoseri, K. N. Al-Khalifa, and A. M. Hamouda, "Re-Thinking Data Strategy and Integration for Artificial Intelligence: Concepts, Opportunities, and Challenges," *Applied Sciences*, vol. 13, p. 7082, 6 2023.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-Decem, pp. 770–778, 12 2016.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), pp. 6000–6010, Curran Associates Inc., 2017.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. J. Burges, L. Bottou, and K. Q. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[12] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[13] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, (New York, NY, USA), pp. 1–12, ACM, 6 2017.

[14] D. Southard, "Tensor Streaming Architecture Delivers Unmatched Performance for Compute Intensive Workloads," 2019.

[15] W. Dai and D. Berleant, "Benchmarking Contemporary Deep Learning Hardware and Frameworks: A Survey of Qualitative Metrics," in *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, pp. 148–155, IEEE, 12 2019.

[16] C. Silvano, D. Ielmini, F. Ferrandi, L. Fiorin, S. Curzel, L. Benini, F. Conti, A. Garofalo, C. Zambelli, E. Calore, S. F. Schifano, M. Palesi, G. Ascia, D. Patti, N. Petra, D. De Caro, L. Lavagno, T. Urso, V. Cardellini, G. C. Cardarilli, R. Birke, and S. Perri, "A Survey on Deep Learning Hardware Accelerators for Heterogeneous HPC Platforms," 6 2023.

[17] V. V. Zunin, "Intel OpenVINO Toolkit for Computer Vision: Object Detection and Semantic Segmentation," in *2021 International Russian Automation Conference (RusAutoCon)*, pp. 847–851, IEEE, 9 2021.

[18] Intel, "OpenVINO: Deep Learning & Vision Tools."

[19] V. Kathail, "Xilinx Vitis Unified Software Platform," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, (New York, NY, USA), pp. 173–174, ACM, 2 2020.

[20] AMD, "Vitis AI: Enabling real-time inference acceleration on AMD Adaptive SoCs, FPGAs and acceleration cards."

[21] E. Jeong, J. Kim, and S. Ha, "TensorRT-Based Framework and Optimization Methodology for Deep Learning Inference on Jetson Boards," *ACM Transactions on Embedded Computing Systems*, vol. 21, pp. 1–26, 9 2022.

[22] Nvidia, "Tensorrt: High performance deep learning inference library."

[23] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized NeuralNetwork Inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, (New York, NY, USA), pp. 65–74, ACM, 2 2017.

[24] S. I. Venieris and C. S. Bouganis, "FpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 2, pp. 326–342, 2019.

[25] M. Hall and V. Betz, "HPIPE: Heterogeneous Layer-Pipelined and Sparse-Aware CNN Inference for FPGAs," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, (New York, NY, USA), pp. 320–320, ACM, 2 2020.

[26] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. Di Guglielmo, P. Harris, J. Krupa, D. Rankin, M. B. Valentin, J. Hester, Y. Luo, J. Mamish, S. Orgrenci-Memik, T. Aarrestad, H. Javed, V. Loncar, M. Pierini, A. A. Pol, S. Summers, J. Duarte, S. Hauck, S.-C. Hsu, J. Ngadiuba, M. Liu, D. Hoang, E. Kreinar, and Z. Wu, "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices," 3 2021.

[27] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 5 2015.

[28] J. Kiefer and J. Wolfowitz, "Stochastic Estimation of the Maximum of a Regression Function," *The Annals of Mathematical Statistics*, vol. 23, pp. 462–466, 9 1952.

[29] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 12 1943.

[30] F. Rosenblatt, "The perceptron - A perceiving and recognizing automaton," Tech. Rep. 85-460-1, Cornell Aeronautical Laboratory, Ithaca, New York, 1 1957.

[31] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, (Cambridge, MA, USA), pp. 2672–2680, MIT Press, 2014.

[32] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[33] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML'10)*, pp. 807–814, 2010.

[34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 10 1986.

[35] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, IEEE, 6 2015.

[36] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," pp. 234–241, 2015.

[37] R. Azad, E. K. Aghdam, A. Rauland, Y. Jia, A. H. Avval, A. Bozorgpour, S. Karimijafarbigloo, J. P. Cohen, E. Adeli, and D. Merhof, "Medical Image Segmentation Review: The Success of U-Net," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, pp. 10076–10095, 12 2024.

[38] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 4 2017.

[39] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3D convolutional networks," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2015 Inter, pp. 4489–4497, 2015.

[40] L. Wang, Y. Xiong, Z. Wang, Y. Qiao, D. Lin, X. Tang, and L. van Gool, "Temporal segment networks: Towards good practices for deep action recognition," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9912 LNCS, pp. 20–36, 2016.

[41] J. Carreira and A. Zisserman, "Quo Vadis, action recognition? A new model and the kinetics dataset," in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 4724–4733, 5 2017.

[42] C. Feichtenhofer, H. Fan, J. Malik, and K. He, "Slowfast networks for video recognition," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2019-Octob, pp. 6201–6210, 2019.

[43] M. Lin, Q. Chen, and S. Yan, "Network In Network," in *2nd International Conference on Learning Representations, ICLR*, (Banff, AB, Canada), 4 2014.

[44] F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1800–1807, IEEE, 7 2017.

[45] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, pp. 834–848, 4 2018.

[46] P. Kumar and N. Singh, "Introduction to Deep Learning with Computer Vision - Types of Convolutions & Atrous Convolutions."

[47] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 9 2018.

[48] J. Hu, L. Shen, S. Albanie, G. Sun, and E. Wu, "Squeeze-and-Excitation Networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 8, pp. 2011–2023, 2020.

[49] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *International Conference on Learning Representations (ICLR)*, (San Diega, CA, USA), 2015.

[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: an imperative style, high-performance deep learning library," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA: Curran Associates Inc., 2019.

[51] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: a system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (USA), pp. 265–283, USENIX Association, 2016.

[52] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe," in *Proceedings of the 22nd ACM international conference on Multimedia*, (New York, NY, USA), pp. 675–678, ACM, 11 2014.

[53] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, (New York, NY, USA), pp. 161–170, ACM, 2 2015.

[54] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A Survey of Accelerator Architectures for Deep Neural Networks," 3 2020.

[55] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks," *Future Internet*, vol. 12, p. 113, 7 2020.

[56] C. Wang and Z. Luo, "A Review of the Optimal Design of Neural Networks Based on FPGA," *Applied Sciences*, vol. 12, p. 10771, 10 2022.

[57] S. I. Venieris, A. Kouris, and C. S. Bouganis, "Toolflows for mapping convolutional neural networks on FPGAS: A survey and future directions," *ACM Computing Surveys*, vol. 51, no. 3, 2018.

[58] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 35–47, 1 2018.

[59] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 10 2016.

[60] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," *Proceedings - IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017*, pp. 152–159, 2017.

[61] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs," *Proceedings - Design Automation Conference*, vol. Part 12828, pp. 3–5, 2017.

[62] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "SCALE-Sim: Systolic CNN Accelerator Simulator," 10 2018.

[63] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, pp. 35–47, 1 2020.

[64] A. Boutros, E. Nurvitadhi, R. Ma, S. Gribok, Z. Zhao, J. C. Hoe, V. Betz, and M. Langhammer, "Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs," in *Proceedings - 2020 International Conference on Field-Programmable Technology, ICFPT 2020*, pp. 10–19, Institute of Electrical and Electronics Engineers Inc., 12 2020.

[65] M. Hall and V. Betz, "From TensorFlow Graphs to LUTs and Wires: Automated Sparse and Physically Aware CNN Hardware Generation," in *Proceedings - 2020 International Conference on Field-Programmable Technology, ICFPT 2020*, pp. 56–65, Institute of Electrical and Electronics Engineers Inc., 12 2020.

[66] A. Montgomerie-Corcoran, Z. Yu, J. Cheng, and C.-S. Bouganis, "PASS: Exploiting Post-Activation Sparsity in Streaming Architectures for CNN Acceleration," 7 2023.

[67] B. Biggs, C.-S. Bouganis, and G. Constantinides, "ATHEENA: A Toolflow for Hardware Early-Exit Network Automation," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 121–132, IEEE, 5 2023.

[68] E. Luo, H. Huang, C. Liu, G. Li, B. Yang, Y. Wang, H. Li, and X. Li, "DeepBurning-MixQ: An Open Source Mixed-Precision Neural Network Accelerator Design Framework for FPGAs," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–9, IEEE, 10 2023.

[69] S. Basalama, A. Sohrabizadeh, J. Wang, L. Guo, and J. Cong, "FlexCNN: An End-to-End Framework for Composing CNN Accelerators on FPGA," *ACM Transactions on Reconfigurable Technology and Systems*, 12 2022.

[70] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 40–47, IEEE, 5 2016.

[71] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Deploying Deep Neural Networks in the Embedded Space," in *2nd International Workshop on Embedded and Mobile Deep Learning (EMDL)*, MobiSys, 6 2018.

[72] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," in *Proceedings of the IEEE*, vol. 75, pp. 1235–1245, 1987.

[73] D. R. Beddiar, B. Nini, M. Sabokrou, and A. Hadid, "Vision-based human activity recognition: a survey," *Multimedia Tools and Applications*, vol. 79, pp. 30509–30555, 11 2020.

[74] I. Jegham, A. Ben Khalifa, I. Alouani, and M. A. Mahjoub, "Vision-based human action recognition: An overview and real world challenges," *Forensic Science International: Digital Investigation*, vol. 32, p. 200901, 2020.

[75] Y. Kong and Y. Fu, "Human Action Recognition and Prediction: A Survey," *International Journal of Computer Vision*, vol. 130, pp. 1366–1401, 5 2022.

[76] G. Anitha and S. Baghavathi Priya, "Posture based health monitoring and unusual behavior recognition system for elderly using dynamic Bayesian network," *Cluster Computing*, vol. 22, pp. 13583–13590, 11 2019.

[77] J. Arunnehru, G. Chamundeeswari, and S. P. Bharathi, "Human Action Recognition using 3D Convolutional Neural Networks with 3D Motion Cuboids in Surveillance Videos," *Procedia Computer Science*, vol. 133, pp. 471–477, 2018.

[78] K. Host and M. Ivašić-Kos, "An overview of Human Action Recognition in sports based on Computer Vision," *Heliyon*, vol. 8, p. e09633, 6 2022.

[79] F. Wu, Q. Wang, J. Bian, N. Ding, F. Lu, J. Cheng, D. Dou, and H. Xiong, "A Survey on Video Action Recognition in Sports: Datasets, Methods and Applications," *IEEE Transactions on Multimedia*, vol. 25, pp. 7943–7966, 2023.

[80] L. Wang, B. Huang, Z. Zhao, Z. Tong, Y. He, Y. Wang, Y. Wang, and Y. Qiao, "VideoMAE V2: Scaling Video Masked Autoencoders with Dual Masking," 3 2023.

[81] Z. Liu, J. Ning, Y. Cao, Y. Wei, Z. Zhang, S. Lin, and H. Hu, "Video Swin Transformer," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3192–3201, IEEE, 6 2022.

[82] K. Li, Y. Wang, Y. He, Y. Li, Y. Wang, L. Wang, and Y. Qiao, "UniFormerV2: Spatiotemporal Learning by Arming Image ViTs with Video UniFormer," 11 2022.

[83] J. Lin, C. Gan, and S. Han, "TSM: Temporal shift module for efficient video understanding," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2019-Octob, pp. 7082–7092, 2019.

[84] C.-L. Zhang, J. Wu, and Y. Li, "ActionFormer: Localizing Moments of Actions with Transformers," pp. 492–510, 2022.

[85] F. Cheng and G. Bertasius, "TallFormer: Temporal Action Localization with a Long-Memory Transformer," pp. 503–521, 2022.

[86] H. Xu, A. Das, and K. Saenko, "R-C3D: Region Convolutional 3D Network for Temporal Activity Detection," in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 5794–5803, IEEE, 10 2017.

[87] R. Hou, C. Chen, and M. Shah, "Tube Convolutional Neural Network (T-CNN) for Action Detection in Videos," in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 5823–5832, IEEE, 10 2017.

[88] J. Pan, S. Chen, M. Z. Shou, Y. Liu, J. Shao, and H. Li, "Actor-Context-Actor Relation Network for Spatio-Temporal Action Localization," in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 464–474, IEEE, 6 2021.

[89] Y. Li, Z. Wang, L. Wang, and G. Wu, "Actions as Moving Points," pp. 68–84, 2020.

[90] X. Yang, X. Yang, M.-Y. Liu, F. Xiao, L. S. Davis, and J. Kautz, "STEP: Spatio-Temporal Progressive Learning for Video Action Detection," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 264–272, IEEE, 6 2019.

[91] Y. Wu, L. Zhu, X. Wang, Y. Yang, and F. Wu, "Learning to Anticipate Egocentric Actions by Imagination," *IEEE Transactions on Image Processing*, vol. 30, pp. 1143–1152, 2021.

[92] A. Furnari and G. Farinella, "What Would You Expect? Anticipating Egocentric Actions With Rolling-Unrolling LSTMs and Modality Attention," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6251–6260, IEEE, 10 2019.

[93] Q. Ke, M. Fritz, and B. Schiele, "Time-Conditioned Action Anticipation in One Shot," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9917–9926, IEEE, 6 2019.

[94] J. Donahue, L. A. Hendricks, M. Rohrbach, S. Venugopalan, S. Guadarrama, K. Saenko, and T. Darrell, "Long-Term Recurrent Convolutional Networks for Visual Recognition and Description," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 677–691, 2017.

[95] J. Y. H. Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici, "Beyond short snippets: Deep networks for video classification," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 07-12-June, pp. 4694–4702, 2015.

[96] C. Yang, Y. Xu, J. Shi, B. Dai, and B. Zhou, "Temporal pyramid network for action recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 588–597, 2020.

[97] D. Tran, H. Wang, L. Torresani, J. Ray, Y. Lecun, and M. Paluri, "A Closer Look at Spatiotemporal Convolutions for Action Recognition," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 6450–6459, 2018.

[98] D. Kondratyuk, L. Yuan, Y. Li, L. Zhang, M. Tan, M. Brown, and B. Gong, "MoViNets: Mobile Video Networks for Efficient Video Recognition," in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 16015–16025, IEEE, 6 2021.

[99] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *ArXiv*, 10 2020.

[100] G. Bertasius, H. Wang, and L. Torresani, "Is Space-Time Attention All You Need for Video Understanding?," 2021.

[101] H. Fan, B. Xiong, K. Mangalam, Y. Li, Z. Yan, J. Malik, and C. Feichtenhofer, "Multiscale Vision Transformers," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6804–6815, IEEE, 10 2021.

[102] A. Piergiovanni, W. Kuo, and A. Angelova, "Rethinking Video ViTs: Sparse Video Tubes for Joint Image and Video Learning," in *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2214–2224, IEEE, 6 2023.

[103] W. Kay, J. Carreira, K. Simonyan, B. Zhang, C. Hillier, S. Vijayanarasimhan, F. Viola, T. Green, T. Back, P. Natsev, M. Suleyman, and A. Zisserman, "The Kinetics Human Action Video Dataset," 5 2017.

[104] R. Goyal, S. E. Kahou, V. Michalski, J. Materzynska, S. Westphal, H. Kim, V. Haenel, I. Fruend, P. Yianilos, M. Mueller-Freitag, F. Hoppe, C. Thurau, I. Bax, and R. Memisevic, "The "Something Something" Video Database for Learning and Evaluating Visual Common Sense," in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 5843–5851, IEEE, 10 2017.

[105] H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre, "HMDB: A large video database for human motion recognition," in *2011 International Conference on Computer Vision*, pp. 2556–2563, IEEE, 11 2011.

[106] M. Monfort, C. Vondrick, A. Oliva, A. Andonian, B. Zhou, K. Ramakrishnan, S. A. Bargal, T. Yan, L. Brown, Q. Fan, and D. Gutfreund, "Moments in Time Dataset: One Million Videos for Event Understanding," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, pp. 502–508, 2 2020.

[107] G. A. Sigurdsson, G. Varol, X. Wang, A. Farhadi, I. Laptev, and A. Gupta, "Hollywood in Homes: Crowdsourcing Data Collection for Activity Understanding," pp. 510–526, 2016.

[108] C. Gu, C. Sun, D. A. Ross, C. Vondrick, C. Pantofaru, Y. Li, S. Vijayanarasimhan, G. Toderici, S. Ricco, R. Sukthankar, C. Schmid, and J. Malik, "AVA: A Video Dataset of Spatio-Temporally Localized Atomic Visual Actions," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6047–6056, IEEE, 6 2018.

[109] H. Fan, X. Niu, Q. Liu, and W. Luk, "F-C3D: FPGA-based 3-dimensional convolutional neural network," in *2017 27th International Conference on Field Programmable Logic and Applications, FPL 2017*, pp. 2–5, 2017.

[110] H. Fan, H. C. Ng, S. Liu, Z. Que, X. Niu, and W. Luk, "Reconfigurable acceleration of 3D-CNNs for human action recognition with block floating-point representation," in *Proceedings - 2018 International Conference on Field-Programmable Logic and Applications, FPL 2018*, no. Section III, pp. 287–294, 2018.

[111] H. Fan, C. Luo, C. Zeng, M. Ferianc, Z. Que, S. Liu, X. Niu, and W. Luk, "F-E3D: FPGA-based acceleration of an efficient 3D convolutional neural network for human action recognition," in *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*, vol. 2019-July, pp. 1–8, 2019.

[112] H. Duan, Y. Zhao, Y. Xiong, W. Liu, and D. Lin, "Omni-Sourced Webly-Supervised Learning for Video Recognition," in *Computer Vision – ECCV 2020. Lecture Notes in Computer Science. Springer*, vol. 12360, pp. 670–688, 2020.

[113] Z. Liu, P. Chow, J. Xu, J. Jiang, Y. Dou, and J. Zhou, "A uniform architecture design for accelerating 2d and 3d cnns on fpgas," *Electronics (Switzerland)*, vol. 8, 1 2019.

[114] S. Winograd, *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, 1 1980.

[115] T. Teng, J. Xi, Z. Letian, W. Xiaotian, W. Jie, and W. Wei, "Exploration of Memory Access Optimization for FPGA-based 3D CNN Accelerator," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1650–1655, EDA Consortium, 2020.

[116] M. Sun, P. Zhao, M. Gungor, M. Pedram, M. Leeser, and X. Lin, "3D CNN acceleration on FPGA using hardware-aware pruning," in *Proceedings - Design Automation Conference*, vol. 2020-July, IEEE, 2020.

[117] F. H. Khan, M. A. Pasha, and S. Masud, "Towards designing a hardware accelerator for 3D convolutional neural networks," *Computers and Electrical Engineering*, vol. 105, p. 108489, 1 2023.

[118] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, 11 1997.

[119] S. I. Venieris and C.-S. Bouganis, "fpgaConvNet: A Toolflow for Mapping Diverse Convolutional Neural Networks on Embedded FPGAs," 11 2017.

[120] C. R. Reeves, ed., *Modern Heuristic Techniques for Combinatorial Problems*. USA: John Wiley & Sons, Inc., 1993.

[121] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and F. F. Li, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1725–1732, IEEE, 6 2014.

[122] MMAction2, "OpenMMLab's Next Generation Video Understanding Toolbox and Benchmark." https://github.com/open-mmlab/mmaction2, 2020.

[123] K. Hara, H. Kataoka, and Y. Satoh, "Can Spatiotemporal 3D CNNs Retrace the History of 2D CNNs and ImageNet?," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 6546–6555, IEEE Computer Society, 12 2018.

[124] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: An efficient hardware accelerator for convolutional neural networks," *Proceedings - IEEE International Symposium on Circuits and Systems*, pp. 0–3, 2017.

[125] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-Decem, pp. 779–788, 6 2016.

[126] Z. Yu and C.-S. Bouganis, "Auto WS: Automate Weights Streaming in Layer-Wise Pipelined DNN Accelerators," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, 3 2024.

[127] L. Petrica, T. Alonso, M. Kroes, N. Fraser, S. Cotofana, and M. Blott, "Memory-Efficient Dataflow Inference for Deep CNNs on FPGA," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, pp. 48–55, IEEE, 12 2020.

[128] T. Alonso, L. Petrica, M. Ruiz, J. Petri-Koenig, Y. Umuroglu, I. Stamelos, E. Koromilas, M. Blott, and K. Vissers, "Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, pp. 1–34, 6 2022.

[129] M. Ibrahim, Z. Zhao, M. Hall, and V. Betz, "Extending Data Flow Architectures for Convolutional Neural Networks to Multiple FPGAs," in *2023 International Conference on Field Programmable Technology (ICFPT)*, pp. 132–141, IEEE, 12 2023.

[130] A. Montgomerie-Corcoran, P. Toupas, Z. Yu, and C.-S. Bouganis, "SATAY: A Streaming Architecture Toolflow for Accelerating YOLO Models on FPGA Devices," in *2023 International Conference on Field Programmable Technology (ICFPT)*, pp. 179–187, IEEE, 12 2023.

[131] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 9 2015.

[132] Y. LeCun, "The MNIST database of handwritten digits," tech. rep., 1998.

[133] Alex Krizhevsky and G. Hinton, "Learning Multiple Layers of Features from Tiny Images," Toronto, ON, Canada, 2009.

[134] H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato, "A Lightweight YOLOv2," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, (New York, NY, USA), pp. 31–40, ACM, 2 2018.

[135] A. Anupreetham, M. Ibrahim, M. Hall, A. Boutros, A. Kuzhively, A. Mohanty, E. Nurvitadhi, V. Betz, Y. Cao, and J.-S. Seo, "High Throughput FPGA-Based Object Detection via Algorithm-Hardware Co-Design," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 17, pp. 1–20, 3 2024.

[136] S. Liu, H. Fan, X. Niu, H.-c. Ng, Y. Chu, and W. LUK, "Optimizing CNN-based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, pp. 1–22, 9 2018.

[137] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, IEEE, 6 2018.

[138] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family," *Proceedings - Design Automation Conference*, vol. 05-09-June, no. October 2017, 2016.

[139] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W. M. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, Institute of Electrical and Electronics Engineers Inc., 11 2018.

[140] S. Winograd, "On multiplication of 2 × 2 matrices," *Linear Algebra and its Applications*, vol. 4, pp. 381–388, 10 1971.

[141] A. Montgomerie-Corcoran, Z. Yu, and C.-S. Bouganis, "SAMO: Optimised Mapping of Convolutional Neural Networks to Streaming Architectures," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 418–424, IEEE, 8 2022.

[142] Glenn Jocher, Ayush Chaurasia, and Jing Qiu, "Ultralytics YOLOv8," 2023.

[143] O. Cicek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation," pp. 424–432, 2016.

[144] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," pp. 740–755, 2014.

[145] B. H. Menze, A. Jakab, S. Bauer, J. Kalpathy-Cramer, K. Farahani, J. Kirby, Y. Burren, N. Porz, J. Slotboom, R. Wiest, L. Lanczi, E. Gerstner, M.-A. Weber, T. Arbel, B. B. Avants, N. Ayache, P. Buendia, D. L. Collins, N. Cordier, J. J. Corso, A. Criminisi, T. Das, H. Delingette, C. Demiralp, C. R. Durst, M. Dojat, S. Doyle, J. Festa, F. Forbes, E. Geremia, B. Glocker, P. Golland, X. Guo, A. Hamamci, K. M. Iftekharuddin, R. Jena, N. M. John, E. Konukoglu, D. Lashkari, J. A. Mariz, R. Meier, S. Pereira, D. Precup, S. J. Price, T. R. Raviv, S. M. S. Reza, M. Ryan, D. Sarikaya, L. Schwartz, H.-C. Shin, J. Shotton, C. A. Silva, N. Sousa, N. K. Subbanna, G. Szekely, T. J. Taylor, O. M. Thomas, N. J. Tustison, G. Unal, F. Vasseur, M. Wintermark, D. H. Ye, L. Zhao, B. Zhao, D. Zikic, M. Prastawa, M. Reyes, and K. Van Leemput, "The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS)," *IEEE Transactions on Medical Imaging*, vol. 34, pp. 1993–2024, 10 2015.

[146] S. Liu and W. Luk, "Towards an Efficient Accelerator for DNN-Based Remote Sensing Image Segmentation on FPGAs," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 187–193, IEEE, 9 2019.

[147] P. Toupas, C.-S. Bouganis, and D. Tzovaras, "fpgaHART: A toolflow for throughput-oriented acceleration of 3D CNNs for HAR onto FPGAs," in *IEEE 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 5 2023.